



# Object-Oriented Component-based Design using Behavioral Contracts: Application to Railway Systems

Sebti Mouelhi, Khalid Agrou, Samir Chouali, Hassan Mountassir

## ► To cite this version:

Sebti Mouelhi, Khalid Agrou, Samir Chouali, Hassan Mountassir. Object-Oriented Component-based Design using Behavioral Contracts: Application to Railway Systems. [Research Report] FEMTO-ST. 2015. hal-01102947v2

**HAL Id: hal-01102947**

**<https://hal.science/hal-01102947v2>**

Submitted on 12 Sep 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



---

INSTITUT FEMTO-ST

UMR CNRS 6174

---

***Object-Oriented Component-Based Design using  
Behavioral Contracts: Application to Railway Systems***

Sebti Mouelhi — Khalid Agrou — Samir Chouali — Hassan Mountassir

---

Rapport de Recherche

DÉPARTEMENT DISC — June 4, 2015



## Object-Oriented Component-Based Design using Behavioral Contracts: Application to Railway Systems

Sebti Mouelhi\*, Khalid Agrou†, Samir Chouali‡, Hassan Mountassir‡

Département DISC

VESONTIO

Rapport de Recherche – June 4, 2015 (23 pages)

**Abstract:** In this report, we propose a formal approach for the design of object-oriented component-based systems using *behavioral contracts*. This formalism merges interface automata describing communication protocols of components with the semantics of their operations. On grounds of consistency with the object-oriented paradigms, we revisit the notions of *incremental design* and *independent implementability* of interface automata by novel definitions of components compatibility, composition, and refinement. Our work is illustrated by a design case study of CBTC railway systems.

**Key-words:** Object-oriented components; Behavioral contracts; Interface automata; Semantics; Refinement; Railway systems.

---

\* SafeRiver, 92120, Montrouge, France (sebti.mouelhi@safe-river.com).

† UPMC Pro, 75005, Paris, France (khalid.agrou@free.fr).

‡ FEMTO-ST Institute, UMR CNRS 6174, 25030, Besançon, France ({schouali,hmountas}@femto-st.fr).



## **Conception par Contrats des Systèmes à base de Composants Orientés Objet: Application aux systèmes Ferroviaires**

**Résumé :** Dans ce rapport, nous proposons une approche formelle pour la conception de systèmes à base de composants orientés objet en utilisant les *contrats de comportement*. Ce formalisme fusionne les *automates d'interface* décrivant les protocoles de communication des composants avec la sémantique de leurs opérations. Pour des raisons de cohérence avec les paradigmes orientés objet, nous revisitons les notions de *conception incrémentale* et l'*implémentation indépendante* établies pour les automates d'interface en définissant autrement la compatibilité, la composition, et le raffinement des composants par le billet de leurs contrats de comportement. Notre travail est illustrée par un cas d'utilisation des systèmes ferroviaires CBTC pour valoir leur pertinence dans le contexte de systèmes critique.

**Mots-clés :** Composants orientés objets; Contrats comportementaux; Automates d'interface; Sémantique; Raffinement; Systèmes ferroviaire.



## 1 Introduction

Component-based development approaches aim to reduce the cost of complex systems design by reusing prefabricated components. A software component is a black box unit of a third-party composition and deployment, with explicit dependencies to its environment [16]. It is exclusively reusable via its interface behavioral specification without disclosing implementation details. However, the design by composition often rises mismatches. A safe interoperability between components shall fulfill two main properties: (1) their interactions do not lead to deadlocks, and (2) the substitution of a component with a new evolved one does not corrupt the compound system operation.

Commonly, the functional interoperability of components is usually checked at the *signature*, *semantic* and *protocol* levels. At the signature level, it is checked on the names and argument types of component operations. At the semantic level, it is verified on the meanings of operations generally modeled by pre/postconditions and invariants. The protocol level regards the scheduling of assumptions on environment inputs to a component, its output guarantees, and its local operations. Protocols can be modeled naturally by *interface automata* [7] obedient to an *optimistic* approach of composition closely related to the object-oriented context: if two or more components communicate within an environment allowing them to avoid deadlocks, they can be used without changes. This approach allows errors detection during the design phase, and hence taking the appropriate decision: either keeping components as they were received from their manufacturer, or requesting their modification.

The first contribution of the report is to demonstrate how object-oriented component-based design (OOCBD) is sound by means of *behavioral contracts* merging interface automata with the semantics of methods. The optimistic approach is accordingly adapted to fulfill the interaction aspects of object-oriented components and the semantic layer of behavioral contracts. In [7, 8], the composition of two interface automata excludes from their synchronized product all states from which the environment cannot prevent deadlocks (semantic and protocol mismatches) by enabling *autonomous* (*output* and *local*) actions. We show that some *input* actions may be also seen as autonomous by regrouping them into *method*, *return*, and *exception* actions.

The second contribution regards *refinement* of behavioral contracts intended to ensure independent implementability of components with respect to their abstract protocol and semantic models. We introduce the *expanding simulation* between interface automata allowing the addition, in a refined behavioral contract of a component, of (i) more services than provided by the abstract one, and (ii) body details (output or internal method calls) for common provided methods with the abstract one. Hence, refinement relation is considered to be *covariant* on input and output actions *i.e.*, refinement issues (resp. provides) more outputs (resp. inputs) than abstraction. An interface automaton  $A'$  refines a second one  $A$  if each input, output, or local event of  $A$  is simulated at least by the same one in  $A'$ . The *alternating simulation* [4], used to refine interface automata in [7, 8], requires *contravariance* on output actions *i.e.*, refinement may issue fewer outputs than abstraction, which is quite inconsistent, from our angle, with the object-oriented context.

All through the report, we justify the relevance of our approach for the design of railway systems. We propose a case study of trains protection functions in modern railway CBTC control systems to track the evolution of safety standards such as the European Norm EN 50128 [2], and launch a new industrial challenge for the design of such critical systems using an object-oriented approach.

This work appeared partially in [6, 14] in preliminary formulations and other contexts. In Section 2, we introduce our case study to not cluttering the contribution sections. It is recalled



gradually to validate the various formal concepts. In sections 3 and 4, we proceed with the study of behavioral contracts, and our approach of components compatibility and composition. Section 5 is devoted to the study of refinement of behavioral contracts. Conclusions are presented in Section 6.

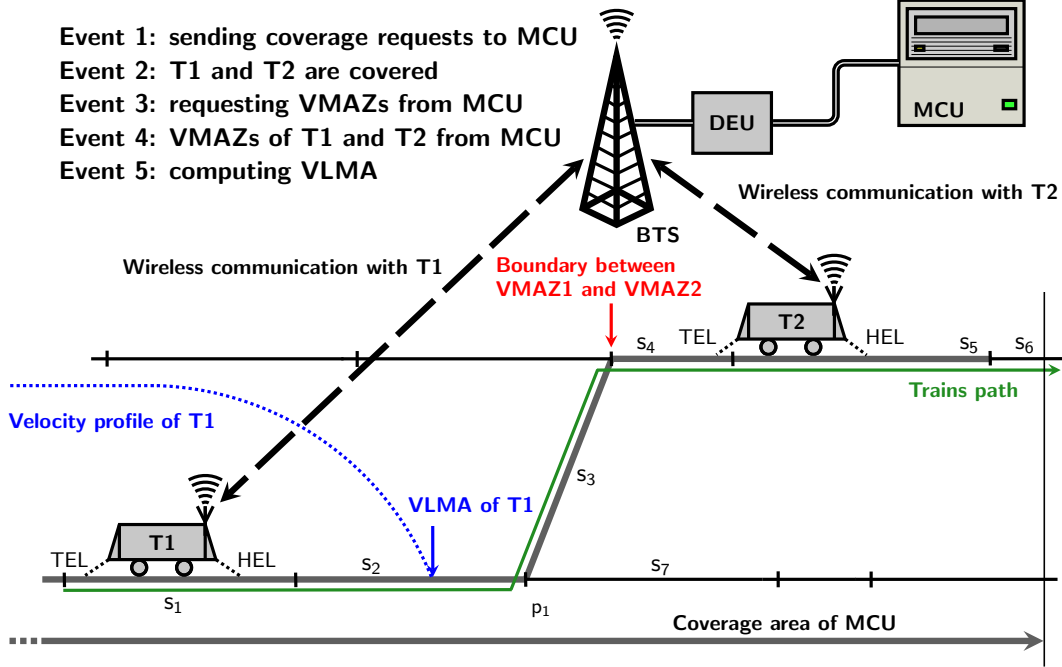


Figure 1: Simplified trains protection in CBTC systems.

## 2 Railway case study

In this section, we introduce a simplified case study of trains protection functions in CBTC (*Communications-Based Train Control*) systems [1], used to illustrate our work (cf. Figure 1). These systems are the next generation of railway control technology which is increasingly being adopted in subways and other similar means of transportation, as well as many industrial major projects worldwide, such as ERTMS/ETCS (*European Rail Traffic Management System/European Train Control System*) [10, 9]. A CBTC system is an automatic train controller independent of track circuits. It determines continuously, in real-time, precise locations of trains, and sends them back control signals by means of bidirectional train-to-wayside data communications. It has train-borne and wayside processor devices implementing automatic train protection (ATP) functions, as well as automatic train operation (ATO), and automatic train supervision (ATS) functions. ATP devices ensure safety-critical requirements (speed control and braking). ATO devices cover non safety-related requirements (doors opening and closing, etc). ATS devices handle the traffic management when necessary [1]. CBTC systems reduce significantly the amount of wayside equipments and allows benefits such as high traffic densities, better headways, reliability of anti-collision processing, adoption of automated trains, etc.

We consider trains control based on *moving block* regime. The positions of a train and its velocity are continuously computed, based on its kinetic and potential energy, and communicated via wireless to wayside equipments. Thus, a *protected area of circulation* is established

for each train up to the next nearest obstacle. The train is consequently able to adapt its speed and braking curves in order to not overcome the limit of this area, called the *danger point* [15].

The *On-Board Device* (OBD) of each train computes two fictional locations: the *tail* and *head external locations* (TEL and HEL). The track fragment between them covers the whole train. Usually, this choice is caught on grounds of safety to keep a safe distance between trains in case of system malfunction. Locations are coordinates on *trains path* composed of *segments* and set in a given direction according to *railroad switches* positions. A segment is identified by a number, a length, and a beginning coordinate. In Figure 1, the switch  $p_1$  is positioned on the segment  $s_3$ , and trains path is the sequence  $s_1, s_2, s_3, s_4, s_5, s_6$ , etc.

The OBDs of T1 and T2 initiate the protection process by asking if they are visible to a *Movement Control Unit* (MCU). There are several MCUs covering the entire line, with overlapping coverage sections allowing safe information handover between them. For simplicity, only one is represented in our case study. Trains locations are sent by wireless to the nearest *Base Transmission Station* (BTS). The latter converts radio signals to digital data and transmits them to the *Data Exchange Unit* (DEU), which in turn transfers them to MCU (event 1). MCU determines whether the zone between TEL and HEL is completely or partially included within its coverage area, and responds T1 and T2. In Figure 1, trains T1 and T2 are both visible to MCU (event 2).

Next, each train asks from its covering MCUs the *Vital Movement Authority Zone* (VMAZ): the area (sequence of segments) in which the train can safely circulate (event 3). In Figure 1, MCU sends to T1 a VMAZ limited by the beginning of  $s_1$  (containing its TEL and HEL) and the end of  $s_3$ , and sends to T2 a VMAZ limited by the beginning of  $s_4$  (containing its TEL) and the end of  $s_5$ , the last segment covered by MCU (event 4). MCU ensures that VMAZs of successive trains never overlap to avoid collisions. VMAZs are computed by chaining segments according to route informations. Chaining may be interrupted up to the nearest obstacle on the train trajectory: the end of MCU coverage area, an uncontrolled switch, or the beginning of the segment containing TEL of the next train, etc. This function is covered by a separate wayside component managing persistent informations (segment and switch locations) and variant ones (switch positions) of the route during the traffic.

Finally, the train computes the danger point, namely the *Vital Limit of Movement Authority* (VLMA), within VMAZ. To locate VLMA, OBD takes a fixed safety margin beforehand the limit of its VMAZ. The train velocity is gradually reduced to reach zero when HEL reaches VLMA (event 5).

### 3 Behavioral Contracts

In this section, we present behavioral contracts combining protocols of object-oriented components, described by interface automata, with the semantics of methods. We start by introducing the formalism of interface automata.

#### 3.1 Interface automata

Interface automata [7] model communication protocols of object-oriented components in terms of temporal scheduling of their *input*, *output*, and *hidden actions*. In OOCBD, input actions may represent component public provided methods, assignment of return values of their calls, and catching their exceptions. Output actions may represent method calls, and return or

exception events. Private methods are implicit and not specified by actions. However, their calls, assignment of their return values, and catching of their thrown exceptions are modeled by hidden actions.

**Definition 1.** An interface automaton  $A$  is a tuple  $(\Upsilon_A, \iota_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A)$  where:  $\Upsilon_A$  is a finite set of states;  $\iota_A \in \Upsilon_A$  is the initial state;  $\Sigma_A^I$ ,  $\Sigma_A^O$ , and  $\Sigma_A^H$  are resp. sets of input, output, and hidden actions;  $\delta_A \subseteq \Upsilon_A \times \Sigma_A \times \Upsilon_A$  is the set of transitions.  $A$  is empty iff  $\Upsilon_A = \emptyset$ .

The alphabet of  $A$  consists of “ $a?$ ” for all  $a \in \Sigma_A^I$ , “ $a!$ ” for all  $a \in \Sigma_A^O$ , and “ $a;$ ” for all  $a \in \Sigma_A^H$ . Sets  $\Sigma_A^{\text{Im}} \subseteq \Sigma_A^I$ ,  $\Sigma_A^{\text{Om}} \subseteq \Sigma_A^O$ , and  $\Sigma_A^{\text{Hm}} \subseteq \Sigma_A^H$ , are resp. actions of public provided methods, call of environment public methods, and calls of private methods. The set  $\Sigma_A^{\text{m}}$  of *method actions* of  $A$  is defined by  $\Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Om}} \cup \Sigma_A^{\text{Hm}}$ .

Given a set of variables  $V$ , we define by  $\mathbb{T}[v]$  the type of  $v \in V$  i.e.,  $v:\mathbb{T}[v]$ , and by  $\mathbb{T}[V] = \prod_{v \in V} \mathbb{T}[v]$  the type of  $V$  (cartesian product of  $\mathbb{T}[v]$  for all  $v \in V$ ). The *signature* of a method action  $a \in \Sigma_A^{\text{m}}$  is  $a(i_1:\mathbb{T}[i_1], \dots, i_k:\mathbb{T}[i_k]) \rightarrow o:\mathbb{T}[o] \# e$ . The set of input parameters of  $a$  is  $\Psi_A^i(a) = \{i_1, \dots, i_k\}$ . The set of return parameters  $\Psi_A^o(a)$  of  $a$  is the singleton  $\{o\}$ . We define  $R_A(a) = o$  the *return action* of  $a$ , and  $E_A(a) = e$  the *exception action* of  $a$ . The set of attributes used by  $a$  is denoted by  $\Lambda_A(a)$  if  $a \in \Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Hm}}$ . The absence of parameters, attributes, or exceptions is represented by a void. If  $R_A(a)$  and  $E_A(a)$  are defined, we set  $\Sigma_A^{\text{r}}$  and  $\Sigma_A^{\text{e}}$  resp. to  $\{R_A(a) \mid a \in \Sigma_A^{\text{m}}\}$  and  $\{E_A(a) \mid a \in \Sigma_A^{\text{m}}\}$ .

We denote, by  $\Sigma_A^{*\text{r}}$  and  $\Sigma_A^{*\text{e}}$ , resp.  $\Sigma_A^{\text{r}} \cap \Sigma_A^{*\text{r}}$  and  $\Sigma_A^{\text{e}} \cap \Sigma_A^{*\text{e}}$  where  $*$   $\in$   $\{I, O, H\}$ . Thus, we deduce obviously that  $\Sigma_A = \Sigma_A^{\text{m}} \cup \Sigma_A^{\text{r}} \cup \Sigma_A^{\text{e}}$ . We set  $\text{Succ}_A(s, a) = t$  such that  $(s, a, t) \in \delta_A$ . A *run*  $\sigma$  of  $A$  is a finite alternated sequence  $s_0[a_0] \dots [a_{n-1}]s_n$  of states and actions where  $(s_k, a_k, s_{k+1}) \in \delta_A$  for all  $k \in \mathbb{N}_{<n}$ . We denote, by  $\Sigma_A\langle\sigma\rangle$ , the set  $\{a_k \in \Sigma_A \mid k \in \mathbb{N}_{<n}\}$  and, by  $\Theta_A(s)$ , the set of runs reaching  $s \in \Upsilon_A$  from  $\iota_A$ . A state  $s \in \Upsilon_A$  is *reachable* in  $A$  if  $\Theta_A(s) \neq \emptyset$ .

*Assumptions:* Interface automata are deterministic, i.e., for all  $(s, a, t_1), (s, a, t_2) \in \delta_A$ ,  $t_1 = t_2$ . All states  $s \in \Upsilon_A$  are reachable in  $A$ . Consider an action  $a \in \Sigma_A^{\text{m}}$  where  $R_A(a)$  and  $E_A(a)$  are defined. If  $a \in \Sigma_A^{\text{Im}}$  (resp.  $\Sigma_A^{\text{Om}}$  and  $\Sigma_A^{\text{Hm}}$ ), then  $E_A(a) \in \Sigma_A^O \setminus \Sigma_A^{\text{m}}$  (resp.  $\Sigma_A^I \setminus \Sigma_A^{\text{m}}$  and  $\Sigma_A^H \setminus \Sigma_A^{\text{m}}$ ): a component providing or requiring  $a$  knows its exception. If  $a \in \Sigma_A^{\text{Im}}$ , then  $R_A(a) \in \Sigma_A^O \setminus \Sigma_A^{\text{m}}$ : the method  $a$  must output its return value. If  $a \in \Sigma_A^{\text{Om}} \cup \Sigma_A^{\text{Hm}}$ , then  $R_A(a)$  may belong or not to  $(\Sigma_A^I \cup \Sigma_A^H) \setminus \Sigma_A^{\text{m}}$ : a component calling  $a$  may require or not the assignment of its return value.

## Well-formedness

Object-oriented implementation rules should be covered by runs of interface automata. A provided public non-void method should be specified at least by a sequence of events starting and ending resp. by an input method action and an output return one, and interposed by calls of local private or environment public methods and assignment of their return values. They may be interleaved optionally by catching or throwing exceptions events. A call of a non-void method, made by a component requiring assignment of its return value, is followed necessarily by a return input action, and optionally by an exception catch one. All actions of a component are *autonomous* (controllable), except method or exception input actions. It's up to the environment to enable or not these actions. In [7, 8], only output and hidden actions are required to be autonomous. From our perspective, input return actions of non-void method calls, made by a component, are also autonomous because the environment is expected to provide their return values and the component has the option to assign them or not.

The set  $\Sigma_A^{\text{aut}}$  of autonomous actions is  $\Sigma_A \setminus (\Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Ie}})$ . We define by  $\Sigma_A^*(s)$  where  $*$   $\in$   $\{I, O, H, \text{Im}, \text{Om}, \text{Hm}, \text{Ir}, \text{Or}, \text{Hr}, \text{Ie}, \text{Oe}, \text{He}, \text{m}, \text{r}, \text{e}, \text{aut}\}$  the set of actions in  $\Sigma_A^*$  enabled from

$s \in \Upsilon_A$ .  $\Sigma_A(s)$  is the set of all enabled actions from  $s$ . The run  $\sigma = s_0[a_0] \dots [a_{n-1}]s_n$  is called autonomous in  $A$  if  $\Sigma_A(\sigma) \subseteq \Sigma_A^{\text{aut}}$  for all  $k \in \mathbb{N}_{<n}$ . It is called *exception-free* if  $\Sigma_A(\sigma) \subseteq \Sigma_A \setminus \Sigma_A^e$  for all  $k \in \mathbb{N}_{<n}$ . A state  $t \in \Upsilon_A$  is reachable autonomously (resp. without exceptions) from  $s \in \Upsilon_A$  in  $A$  if there is an autonomous (resp. exception-free) run between  $s$  and  $t$ .

**Definition 2.** An interface automaton  $A$  is well-formed iff for all  $s \in \Upsilon_A$ , and  $a \in \Sigma_A^{\text{m}}(s)$  where  $R_A(a) \in \Sigma_A^{\text{r}}$ , there is at least a state  $t \in \Upsilon_A$ , where  $R_A(a) \in \Sigma_A^{\text{r}}(t)$ , reachable autonomously without exceptions from  $\text{Succ}_A(s, a)$ .

### 3.2 Method semantics

The semantics of a provided method consists of: (i) a precondition representing environment assumptions on input parameters, (ii) an abstract specification of the return parameter computation using input parameters and attributes, (iii) a termination postcondition on the return parameter depending on input parameters and attributes, and (iv) an extra postcondition describing exception conditions on parameters and attributes. A method call semantics is defined only by a precondition on input parameters and a postcondition on input and return parameters.

Given a set of variables  $V$ , a *condition* on  $v$  is a subtype of  $\mathbb{T}[v]$ . A condition  $Q$  on  $V$  is a subtype of  $\mathbb{T}[V]$ . We denote by  $Q[w_1, \dots, w_n]$  (or  $Q[W]$ ), the *projection* of  $Q$  on variables in  $W = \{w_1, \dots, w_n\} \subseteq V$ . These conditions can be concretely defined as predicates in theories adapted to the variable types. Consider the set  $Z \subseteq W$ , and two conditions  $P$  and  $Q$  subtypes of  $\mathbb{T}[V]$ , we set the following equivalences to define semantic formulas in the rest of the report:

- $\perp[W] \equiv P[W] = \emptyset$ ;  $\top[W] \equiv P[W] = \mathbb{T}[W]$ ;
- $\neg P[W] \equiv \mathbb{T}[W] \setminus P[W]$ ;
- $P[Z] \wedge Q[W] \equiv (P[Z] \times Q[W \setminus Z]) \cap Q[W]$ ;
- $P[Z] \vee Q[W] \equiv (P[Z] \times Q[W \setminus Z]) \cup Q[W]$ ;
- $P[W] \Rightarrow Q[W] \equiv P[W] \subseteq Q[W]$ .

**Definition 3.** Given an interface automaton  $A$ , an input semantics  $I_a = (P_a, S_a, Q_a, E_a)$  of an action  $a \in \Sigma_A^{\text{Im}}$  is defined by:

- a precondition  $P_a \subseteq \mathbb{T}[\Psi_A^{\text{i}}(a)]$ ;
- a specification  $S_a \subseteq \mathbb{T}[\Psi_A^{\text{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\text{o}}(a)]$ ;
- a termination postcondition  $Q_a \subseteq \mathbb{T}[\Psi_A^{\text{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\text{o}}(a)]$ ;
- an exception postcondition  $E_a \subseteq \mathbb{T}[\Psi_A^{\text{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\text{o}}(a)]$ .

An output semantics  $O_b = (P_b, Q_b)$  of an action  $b \in \Sigma_A^{\text{Om}}$  is defined by:

- a precondition  $P_b \subseteq \mathbb{T}[\Psi_A^{\text{i}}(b)]$ ;
- a postcondition  $Q_b \subseteq \mathbb{T}[\Psi_A^{\text{i}}(b) \cup \Psi_A^{\text{o}}(b)]$ .

These conditions are denoted resp. by  $I_a.P$ ,  $I_a.S$ ,  $I_a.Q$ ,  $I_a.E$ ,  $O_b.P$ , and  $O_b.Q$ .

In the previous definition, we consider only the semantics of observable method actions ( $a \in \Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Om}}$ ). We omit the semantics of private method actions ( $a \in \Sigma_A^{\text{Hm}}$ ) because they are not relevant for interoperability. We define behavioral contracts as follows.

**Definition 4.** A behavioral contract  $B$  of a component is a tuple  $(A, \mathcal{I}, \mathcal{O})$  such that  $A$  is an interface automaton,  $\mathcal{I}$  is a map associating each  $a \in \Sigma_A^{\text{Im}}$  to an input semantics  $I_a$ , and  $\mathcal{O}$  is a map associating each  $a \in \Sigma_A^{\text{Om}}$  to an output semantics  $O_a$ . We denote by,  $B.A$ , the interface automaton of  $B$ ,  $B.\mathcal{I}$ , the map  $\mathcal{I}$  of  $B$ , and  $B.\mathcal{O}$ , the map  $\mathcal{O}$  of  $B$ .

**Definition 5.** Given a behavioral contract  $B$  and an action  $a \in \Sigma_A^{\text{Im}}$  where  $B.A = A$  and  $B.I(a) = (P_a, S_a, Q_a, E_a)$ , for all  $(i, f, o) \in \Psi_A^i(a) \times \Lambda_A(a) \times \Psi_A^o(a)$ ,

- $a$  is correct with respect to  $B.I(a)$  iff  $P_a[i] \wedge S_a[i, f, o] \Rightarrow Q_a[i, f, o]$ ;
- $a$  throws exceptions with respect to  $B.I(a)$  iff  $P_a[i] \wedge S_a[i, f, o] \Rightarrow E_a[i, f, o]$ ;
- $a$  terminates with respect to  $B.I(a)$  iff  $P_a[i] \wedge S_a[i, f, o] \Rightarrow Q_a[i, f, o] \wedge \neg E_a[i, f, o]$ .

The previous definition establishes the different relations between the specification and pre/postconditions of an input method action  $a \in \Sigma_A^{\text{Im}}$ . Stated conditions are based on the Hoare triplet [11]: a provided method is correct if its behavior under the precondition ensures the termination postcondition, it throws exceptions if the exception postcondition is satisfied, and terminates if it is correct and the exception postcondition is not satisfied.

### 3.3 Design of the railway case study

The UML-like component architecture in Figure 2 represents ATP protection equipments mentioned in Section 2. We count four component classes: OnBoardDevice, DataExchangeUnit, MovementControlUnit, and SubRouteBuilder instantiated resp. by components OBD, DEU, MCU, and SRB. The last three ones implement resp. interfaces DataExchange, MovementControl, and RouteBuilder.

The component DEU implements the public (+) method *covReq* (coverage request), whose arguments are: *tel* and *ts*, resp. the coordinate of TEL and the identifier of the segment containing TEL, *hel* and *hs*, resp. the coordinate of HEL and the identifier of the segment containing HEL, and *t*, the train identifier. According to the interface automaton  $A_d$  of DEU (cf. Figure 3(b)), the method *covReq* transfers the coverage request to MCU by invoking the method *isCovered*. MCU responds OBD, via DEU, by returning 2 (resp. 1) if it covers completely (resp. partially) the train (signal *covered*), or by throwing *uncovered* if not.

Subsequently, if the train is covered by MCU, OBD requests its VMAZ (*vmazReq*). DEU transfers the request by calling *computeVmaz* implemented by MCU. In turn, MCU calls the method *chain* of SRB to perform chaining on segments in order to compute VMAZ bounds within the sequence of segments from *start* to *end* (arguments of *chain*). If MCU covers only HEL, *start* is set to the first segment in trains path fully covered by MCU. It is set to *ts* otherwise. The argument *end* is set to the segment containing the next obstacle. According to  $A_m$  in Figure 3(c), if chaining is interrupted by an uncontrolled switch, MCU handles the exception *uncontSW* expected to be thrown by *chain* and in turn, throws *default*.

Based on the path database *bdd*, SRB returns segments of VMAZ in the table *segs* of size *max*, the maximum number of segments covered by MCU. The field *useful\_nb*  $\leq$  *max* indicates the number of segments included in VMAZ. MCU computes accordingly the VMAZ bounds coordinates on the path frame based on informations of useful segments (identifiers, beginning coordinates, and lengths saved in data structures of type *Seg*). In the case where MCU covers only a part of the train VMAZ, it returns a pair *vmaz* of coordinates where one of them is null and the other is a positive real. Otherwise, both of them are positive reals. The map attribute *cst* (covered segments and trains) is finally updated such that segments covered both by MCU and VMAZ of the train are associated to its identifier.

According to  $A_o$  in Figure 3(a), OBD fixes VLMA by calling its private (−) method *computeVlma* within VMAZ. Consequently, it controls the train speed if HEL is sufficiently far from VLMA (*ctrlVelocity*), or performs an emergency brake (*emgcyBrake*) otherwise.

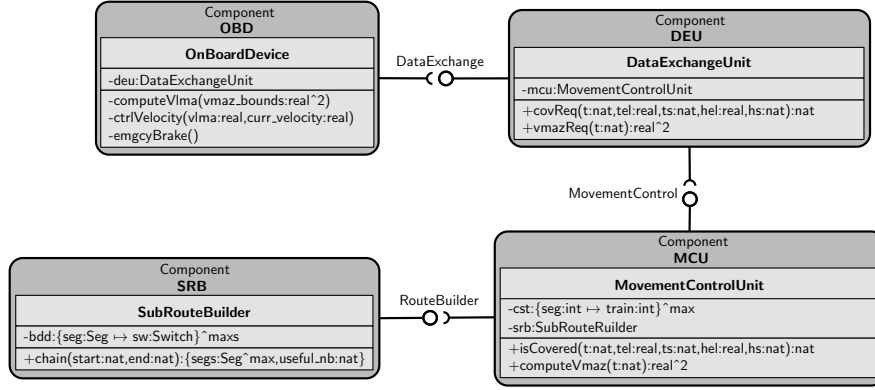


Figure 2: UML-like component architecture.

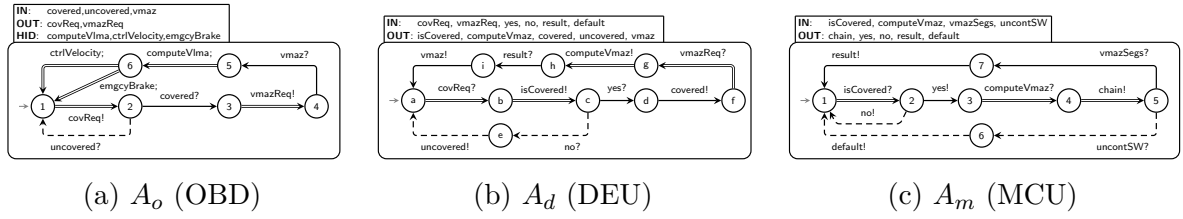


Figure 3: Interface automata of OBD, DEU, and MCU: method actions (double transitions); return actions (simple transitions); exception actions (dashed transitions).

Table 1: Semantics of the method action *covReq*.

| Output semantics $B_o.\mathcal{O}$   | Input semantics $B_d.\mathcal{I}$   |
|--|---|
| $B_o.\mathcal{O}(covReq).P \equiv t \in \{0, \dots, 30\} \wedge$<br>$ts, hs \in \{0, \dots, 500\} \wedge$<br>$tel, hel \in [0, 5000] \wedge tel < hel$ | $B_d.\mathcal{I}(covReq).P \equiv t \in \{0, \dots, 30\} \wedge$<br>$ts, hs \in \{0, \dots, 500\} \wedge$<br>$tel, hel \in [0, 5000]$ |
| $B_o.\mathcal{O}(covReq).Q \equiv covered \in \{0, 1, 2\}$   | $B_d.\mathcal{I}(covReq).S \equiv \perp[t, tel, hel, ts, hs, covered]$  |
|  | $B_d.\mathcal{I}(covReq).Q \equiv covered \in \{1, 2\}$   |
|  | $B_d.\mathcal{I}(covReq).E \equiv covered = 0$  |

Let us consider three behavioral contracts  $B_o$ ,  $B_d$ , and  $B_m$  resp. for components OBD, DEU, and MCU where  $B_o.\mathcal{A}$  is  $A_o$ ,  $B_d.\mathcal{A}$  is  $A_d$ , and  $B_m.\mathcal{A}$  is  $A_m$ . Table 1 shows an example of the semantics of the method *covReq* in  $B_o$  and  $B_d$  whose signature is *covReq* ( $t, tel, ts, hel, hs$ )  $\rightarrow$  *covered*  $\sharp$  *uncovered* (parameter types are given in Figure 2). The semantics of *covReq* in  $B_o$  and  $B_d$  states that minimal and maximal identifiers  $t$  of trains, are resp. 0 and 30, and those of segment identifiers ( $ts$  and  $hs$ ), are resp. 0 and 500. The precondition of *covReq* in  $B_o$  states that conditions  $tel, hel \in [0, 5000]$  and  $tel < hel$  have to be satisfied by calling the method, where 5000um (unit of measurement) is the size of the longest trains path. In  $B_d$ , the precondition states simply that  $tel, hel \in [0, 5000]$ . In  $B_o$ , the termination postcondition of *covReq* states that the return parameter *covered* is a signal in  $\{0, 1, 2\}$ . However, it states only that *covered* is a signal in  $\{1, 2\}$  in  $B_d$ , because if it is equal to 0, the exception *uncovered* is thrown. The specification  $B_d.\mathcal{I}(covReq).S$  is not defined ( $\perp[t, tel, hel, ts, hs, covered]$ ): at the level of  $B_d$ , there is no parameter or attribute ( $\Lambda_{A_d}(covReq) = \emptyset$ ) describing how the return parameter *covered* is computed. MCU, after receiving the coverage request from OBD, is expected to ask SRB to check in *bdd* whether  $tel$  and  $hel$  are really placed resp. on  $ts$  and  $hs$ , as claimed by OBD. This

functionality of SRB does not appear intentionally at this stage. We expect using this detail to justify our refinement approach presented in Section 5.

Finally, by considering that  $R_{A_o}(covReq)$  is *covered*  $\in \Sigma_{A_o}^r$ ,  $R_{A_o}(vmazReq)$  is *vmaz*  $\in \Sigma_{A_o}^r$ , and  $E_{A_o}(covReq)$  is *uncovered*  $\in \Sigma_{A_o}^e$ , we deduce that  $A_o$  is well-formed. The reader can easily deduce the well-formedness of  $A_d$  and  $A_m$  by finding their method, return and exception actions.

## 4 Components Composition

The composition of two behavioral contracts may induce deadlock situations caused by potential semantic or protocol incompatibilities. At the protocol level, the composition of two interface automata may contain *deadlock* states from which one of them requests an input not accepted by the other. For example, a component calls a method throwing exceptions without handling them. In Java, a deadlock is the detection of a method call exception not included in a clause try/catch. The thrown exception is the output action and the try/catch freedom account for the absence of the corresponding input action.

At the semantic level, the synchronization of shared input/output method actions with incompatible semantics, leads to deadlock states. A component outputting a method call have more informations about its arguments. Thus, the call precondition is stronger than that of the method implementation: the environment is expected to provide input arguments included in the implementation precondition. In return, the component providing the method communicates to the environment a postcondition on its return parameter: it vouches to provide only return values that satisfy the postcondition. The calling component cannot have more detailed informations about the return parameter than the implementing one. That's why the postcondition of a method invocation is weaker than that provided by its implementation. Note that preconditions, like postconditions, of provided observable methods are required to be satisfiable. Not all calling environments satisfy the precondition, or expect return guarantees larger than the postcondition [8]. In this case, synchronization disparities are detected.

### 4.1 Synchronization of interface automata and semantic compatibility

The synchronization of two interface automata  $A_1$  and  $A_2$  is possible only if they are mutually composable *i.e.*,  $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_2}^H \cap \Sigma_{A_1} = \emptyset$ . The set of shared input/output actions in  $A_1$  and  $A_2$  is  $Shared(A_1, A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ . For simplicity, we denote the couple of states  $(s_1, s_2)$  by  $s_1s_2$ . By synchronizing  $A_1$  and  $A_2$ , transitions labeled by shared actions synchronize and the others are interleaved.

The *synchronized product*  $A_1 \otimes A_2$  of  $A_1$  and  $A_2$  is an interface automaton where  $\Upsilon_{A_1 \otimes A_2} = \Upsilon_{A_1} \times \Upsilon_{A_2}$ ,  $\iota_{A_1 \otimes A_2} = \iota_{A_1} \iota_{A_2}$ ,  $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \setminus Shared(A_1, A_2)$ ,  $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \setminus Shared(A_1, A_2)$ ,  $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H \cup Shared(A_1, A_2)$ , and  $(s_1s_2, a, t_1t_2) \in \delta_{A_1 \otimes A_2}$  iff :

- $a \in Shared(A_1, A_2) \wedge (s_1, a, t_1) \in \delta_{A_1} \wedge (s_2, a, t_2) \in \delta_{A_2}$ ;
- $a \notin Shared(A_1, A_2) \wedge ((s_1, a, t_1) \in \delta_{A_1} \wedge s_2 = t_2 \vee (s_2, a, t_2) \in \delta_{A_2} \wedge s_1 = t_1)$ .

Given tow behavioral contracts  $B_1$  and  $B_2$  where  $B_1.\mathcal{A} = A_1$  and  $B_2.\mathcal{A} = A_2$ ,  $B_1$  and  $B_2$  are *composable* if  $A_1$  and  $A_2$  are composable, and each  $a \in Shared(A_1, A_2) \cap \Sigma_{A_1}^m$  has the same signature in  $A_1$  and  $A_2$ . We deduce, from the composability of  $B_1$  and  $B_2$ , that for each  $a \in \Sigma_{A_1}^m \cap Shared(A_1, A_2)$ , if  $R_{A_i}(a), E_{A_i}(a) \in \Sigma_{A_i}$  for all  $i \in \{1, 2\}$ , then  $R_{A_1}(a) = R_{A_2}(a) = r_a$ ,  $E_{A_1}(a) = E_{A_2}(a) = e_a$  and  $r_a, e_a \in Shared(A_1, A_2)$ . In the following definition, we provide semantic compatibility conditions on input/output method actions shared between  $A_1$  and  $A_2$ .

**Definition 6.** Given  $a \in \text{Shared}(A_1, A_2) \cap \Sigma_{A_1}^m$ , for all  $(i, o) \in \Psi_{A_1}^i(a) \times \Psi_{A_1}^o(a)$ , if one of the following conditions holds, then  $a$  in  $B_1$  is semantically compatible with  $a$  in  $B_2$  i.e.,  $\text{SemComp}_a(B_1, B_2)$ :

- $B_1.\mathcal{O}(a).P[i] \Rightarrow B_2.\mathcal{I}(a).P[i] \wedge B_1.\mathcal{O}(a).Q[i, o] \Leftarrow B_2.\mathcal{I}(a).Q[i, o]$ , if  $a \in \Sigma_{A_1}^{Om}$ ;
- $B_1.\mathcal{I}(a).P[i] \Leftarrow B_2.\mathcal{O}(a).P[i] \wedge B_1.\mathcal{I}(a).Q[i, o] \Rightarrow B_2.\mathcal{O}(a).Q[i, o]$ , if  $a \in \Sigma_{A_1}^{Im}$ .

**Example 1.** According to our case study (cf. Section 3.3),  $B_o.\mathcal{A}$  and  $B_d.\mathcal{A}$  are composable. The set  $\text{Shared}(A_o, A_d)$  is defined by  $\{\text{covReq}, \text{vmazReq}, \text{covered}, \text{uncovered}, \text{vmaz}\}$ . Based on Table 1,  $\text{SemComp}_a(B_o, B_d)$  is true for  $a = \text{covReq}$ .

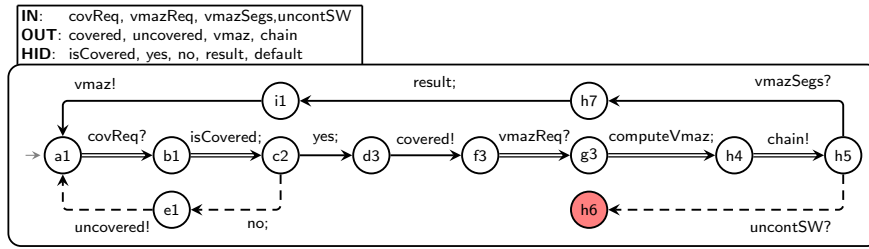


Figure 4: Interface automaton  $(B_d|B_m).\mathcal{A}$ .

**Definition 7.** Assume that  $B_1$  and  $B_2$  are composable, we define by  $B_1|B_2$ , the synchronized behavioral contract of  $B_1$  and  $B_2$  where:

- $(B_1|B_2).\mathcal{A}$  is defined by  $A_1 \otimes A_2$  restricted to the set of reachable states from  $\iota_{A_{12}}$ ;
- $(B_1|B_2).\mathcal{I}$  is defined by:
  - $B_1.\mathcal{I}(a)$  for all  $a \in \Sigma_{A_1}^{Im} \setminus \text{Shared}(A_1, A_2)$ ;
  - $B_2.\mathcal{I}(a)$  for all  $a \in \Sigma_{A_2}^{Im} \setminus \text{Shared}(A_1, A_2)$ ;
- $(B_1|B_2).\mathcal{O}$  is defined by:
  - $B_1.\mathcal{O}(a)$  for all  $a \in \Sigma_{A_1}^{Om} \setminus \text{Shared}(A_1, A_2)$ ;
  - $B_2.\mathcal{O}(a)$  for all  $a \in \Sigma_{A_2}^{Om} \setminus \text{Shared}(A_1, A_2)$ .

We denote  $(B_1|B_2).\mathcal{A}$  by  $A_{12}$  for simplicity. Deadlock states in  $A_{12}$  represent possible deadlocks during the communication between components specified by  $B_1$  and  $B_2$  at the protocol and semantic levels. They are states  $s_1s_2$  such that (i) there exists at least  $a \in \text{Shared}(A_1, A_2)$  enabled from  $s_1$  and not from  $s_2$  or inversely, or (ii)  $a$  is a method action enabled from  $s_1$  and  $s_2$  but, the condition  $\text{SemComp}_a(B_1, B_2)$  is not satisfied.

The latter condition on semantics compatibility requires that the calling component shall define properly the output semantics of the method call with respect to the input semantics imposed by its environment: this allows the detection of the assumptions on components exchanged data as early as possible, and make the design more reliable.

**Definition 8.** The set of deadlock states  $\text{Dead}(A_1, A_2)$  in  $A_{12}$  is  $\{s_1s_2 \in \Upsilon_{A_{12}} \mid (\exists a \in \text{Shared}(A_1, A_2). D_1(s_1s_2) \vee D_2(s_1s_2))\}$  where

$$D_1(s_1s_2) \equiv (a \in \Sigma_{A_1}^O(s_1) \wedge a \notin \Sigma_{A_2}^I(s_2)) \vee (a \in \Sigma_{A_1}^{Om}(s_1) \wedge a \in \Sigma_{A_2}^{Im}(s_2) \wedge \neg \text{SemComp}_a(B_1, B_2));$$

$$D_2(s_1s_2) \equiv (a \in \Sigma_{A_2}^O(s_2) \wedge a \notin \Sigma_{A_1}^I(s_1)) \vee (a \in \Sigma_{A_2}^{Om}(s_2) \wedge a \in \Sigma_{A_1}^{Im}(s_1) \wedge \neg \text{SemComp}_a(B_1, B_2)).$$



**Example 2.** Interface automata  $A_d$  and  $A_m$  are composable (cf. Figure 3). Let us consider two composable behavioral contracts  $B_d$  and  $B_m$  where  $B_d.\mathcal{A} = A_d$  and  $B_m.\mathcal{A} = A_m$ . By supposing that actions *isCovered* and *computeVmaz* are semantically compatible between  $B_d$  and  $B_m$ , the state  $h6$  is the only deadlock state in  $(B_d|B_m).\mathcal{A}$ : the exception action *default*  $\in \Sigma_{A_m}^e(6) \cap \text{Shared}(A_d, A_m)$  is not enabled from the state  $h$  in  $A_d$  (cf. Figure 4).

## 4.2 Optimistic approach of composition

The incremental bottom-up design means that the compatibility checking between components can be performed for partial descriptions of the system. The optimistic approach of interface automata composition is closely consistent with the incremental design oncoming.

In this approach, the presence of deadlock states in  $A_{12}$  doesn't imply necessarily that  $B_1$  and  $B_2$  are incompatible: the existence of a suitable environment  $E$  providing good input steps and semantics for  $A_{12}$  such that deadlock states are prevented, implies that they are *compatible*.  $E$  must satisfy the following conditions: (1)  $E$  and  $B_1|B_2$  are composable, (2)  $E.\mathcal{A}$  is non-empty interface automaton, (3)  $\text{Dead}(A_{12}, E.\mathcal{A}) = \emptyset$ , and (4) no state in the set  $\text{Dead}(A_1, A_2) \times \Upsilon_{E.\mathcal{A}}$  is reachable in  $((B_1|B_2)|E).\mathcal{A}$  [7].

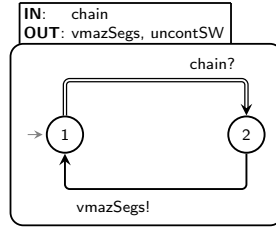
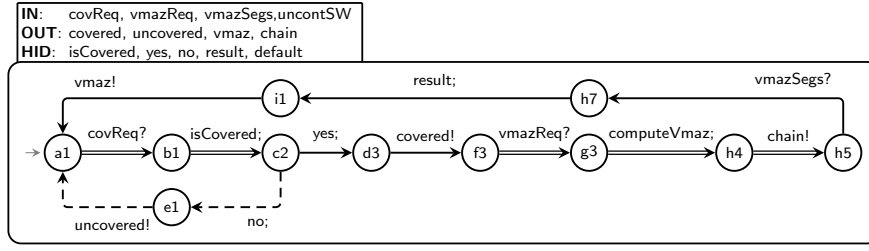


Figure 5: Interface automaton  $A_s$  of SRB.

**Example 3.** We assume that SRB does not throw the exception *uncontSW* if an uncontrolled switch is met during chaining. The train VAMZ is limited by the switch position: for example, if  $p_1$  is uncontrolled, VMAZ of T1 is bounded by the end of segment  $s_2$  (cf. Figure 1). Let us consider a behavioral contract  $B_s$  for SRB composable with  $B_d|B_m$  where  $B_s.\mathcal{A} = A_s$  (cf. Figure 5) and the condition  $\text{SemComp}_a(B_d|B_m, B_s)$  is valid for  $a = \text{chain}$ .  $B_s$  is a suitable environment for  $B_d|B_m$ . In  $((B_d|B_m)|B_s).\mathcal{A}$ , states  $h61$  and  $h62$  are not reachable, because from the state 2 in  $A_s$ , the action *uncontSW* is not enabled. Consequently,  $B_d$  and  $B_m$  are compatible behavioral contracts.

In the product  $A_{12}$ , all states  $s_1s_2$  from which deadlock states are autonomously reachable (cf. Section 3.1), are considered as incompatible and must be removed from  $A_{12}$ . No environment can prevent reaching deadlocks from these states. A state  $s_1s_2 \in \Upsilon_{A_{12}}$  is *compatible* in  $A_{12}$  if there is no state  $d_1d_2 \in \text{Dead}(A_1, A_2)$  autonomously reachable from  $s_1s_2$ . We denote, by  $\text{Cmp}(A_1, A_2)$ , the set of compatible states in  $A_{12}$ .  $B_1$  and  $B_2$  are *compatible* iff they are composable and  $\imath_{A_{12}} \in \text{Cmp}(A_1, A_2)$ . The interface automaton of the composition of two behavioral contracts is restricted to compatible states of their synchronized product.

**Definition 9.** The composition  $B_1||B_2$  of two compatible behavioral contracts  $B_1$  and  $B_2$  is defined by  $(B_1||B_2).\mathcal{I} = (B_1|B_2).\mathcal{I}$ ,  $(B_1||B_2).\mathcal{O} = (B_1|B_2).\mathcal{O}$ , and  $(B_1||B_2).\mathcal{A}$  is an interface automaton where  $\Upsilon_{(B_1||B_2).\mathcal{A}}$  is restricted to  $\text{Cmp}(A_1, A_2)$ ,  $\imath_{(B_1||B_2).\mathcal{A}} = \imath_{A_{12}}$ ,  $\Sigma_{(B_1||B_2).\mathcal{A}}^* = \Sigma_{A_{12}}^*$  for  $*$   $\in \{\text{I}, \text{O}, \text{H}\}$ , and  $\delta_{(B_1||B_2).\mathcal{A}} = \{(s, a, t) \in \delta_{A_{12}} \mid s, t \in \text{Cmp}(A_1, A_2)\}$ .

Figure 6: Interface automaton  $(B_d || B_m).A$ .

**Example 4.** The interface automaton  $(B_d || B_m).A$  is the restriction of  $(B_d || B_m).A$  to the set  $\Upsilon_{(B_d || B_m).A} \setminus \{h6\}$  of compatible states (cf. Figure 6). Assume that  $A_d$  is not well-formed and do not expect to assign the return value of *computeVmaz*,  $h7$  is a deadlock state in  $(B_d || B_m).A$ . In this case, states  $h5$ ,  $h4$ , and  $g3$  are incompatible (the path between  $g3$  and  $h7$  is autonomous): the call of *vmazReq* leads inevitably to deadlocks for all possible environments.

The following property states the preservation of interface automata well-formedness by composition of behavioral contracts.

**Theorem 1.** *If  $B_1$  is compatible with  $B_2$  and  $B_1.A$  and  $B_2.A$  are well-formed, then  $(B_1 || B_2).A$  is also well-formed.*

*Proof.* We denote  $(B_1 || B_2).A$  by  $A'_{12}$ . Given  $s_1 s_2 \in \Upsilon_{A'_{12}}$  and  $a \in \Sigma_{A'_{12}}^m(s_1 s_2)$  where  $R_{A'_{12}}(a) \in \Sigma_{A'_{12}}^r$ , we have to prove that, there is at least  $t_1 t_2 \in \Upsilon_{A'_{12}}$ , where  $R_{A'_{12}}(a) \in \Sigma_{A'_{12}}^r(t_1 t_2)$ , reachable from  $Succ_{A'_{12}}(s_1 s_2, a)$  by an autonomous exception-free run? We have the following assumptions: (i) if  $a \in \Sigma_{A_1}^m(s_1)$  and  $R_{A_1}(a) \in \Sigma_{A_1}^r$ , then there is at least a state  $t_1 \in \Upsilon_{A_1}$  such that  $R_{A_1}(a) \in \Sigma_{A_1}^r(t_1)$  reachable from  $Succ_{A_1}(s_1, a)$  by an autonomous exception-free run  $\sigma_1 = s_1^1[a_1^1] \dots s_1^{k-1}[a_1^{k-1}]s_1^k$  where  $s_1^1 = Succ_{A_1}(s_1, a)$  and  $s_1^k = t_1$ ; (ii) if  $a \in \Sigma_{A_2}^m(s_2)$  and  $R_{A_2}(a) \in \Sigma_{A_2}^r$ , then there is at least a state  $t_2 \in \Upsilon_{A_2}$  such that  $R_{A_2}(a) \in \Sigma_{A_2}^r(t_2)$  reachable from  $Succ_{A_2}(s_2, a)$  by an autonomous exception-free run  $\sigma_2 = s_2^1[a_2^1] \dots s_2^{l-1}[a_2^{l-1}]s_2^l$  where  $s_2^1 = Succ_{A_2}(s_2, a)$  and  $s_2^l = t_2$ .

(1) If  $a \in \Sigma_{A_1}^m \cap Shared(A_1, A_2)$ , we have  $R_{A_1}(a) = R_{A_2}(a) = r_a$  since  $B_1$  and  $B_2$  are composable. The transitions labeled by  $a$  enabled from  $s_1$  and  $s_2$  synchronize if  $SemComp_a(A_1, A_2)$ :

(1.1) if  $r_a \in Shared(A_1, A_2)$ , then the transitions enabled from  $s_1^k$  and  $s_2^l$  labeled by  $r_a$  synchronize. If  $(\Sigma_{A_1} \langle \sigma_1 \rangle \cup \Sigma_{A_2} \langle \sigma_2 \rangle) \cap Shared(A_1, A_2) = \emptyset$ , then the transitions of  $\sigma_1$  and  $\sigma_2$  are interleaved asynchronously and produce autonomous exception-free runs between  $s_1^1 s_2^1$  and  $s_1^k s_2^l$ . If  $(\Sigma_{A_1} \langle \sigma_1 \rangle \cup \Sigma_{A_2} \langle \sigma_2 \rangle) \cap Shared(A_1, A_2) \neq \emptyset$ , then all transitions labeled by shared actions in  $\sigma_1$  and  $\sigma_2$  synchronize and produce autonomous exception-free runs between  $s_1^1 s_2^1$  and  $s_1^k s_2^l$ . For each  $\sigma$  from those runs, if  $\Upsilon_{A_{12}} \langle \sigma \rangle \cup \{Succ_{A_{12}}(s_1^k s_2^l, r_a)\} \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  if  $s_1 s_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $s_1 s_2 \notin \Upsilon_{A'_{12}}$ .

(1.2) if  $r_a \notin Shared(A_1, A_2)$  and  $a \in \Sigma_{A_1}^m$ , then the transition enabling  $r_a$  as output action is interleaved form  $s_1^k t_2$  where  $t_2$  is reachable from  $s_2$  in  $A_2$ . If  $\Sigma_{A_1} \langle \sigma_1 \rangle \cap Shared(A_1, A_2) = \emptyset$ , the transitions of  $\sigma_1$  are interleaved and among the produced runs, we distinguish the autonomous exception-free run  $\sigma = s_1^1 t_2[a_1^1] \dots s_1^{k-1} t_2[a_1^{k-1}]s_1^k t_2$  in  $A_{12}$  where  $t_2 = s_2^1$ : if  $\Upsilon_{A_{12}} \langle \sigma \rangle \cup \{Succ_{A_{12}}(s_1^k t_2, r_a)\} \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  if  $s_1 s_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $s_1 s_2 \notin \Upsilon_{A'_{12}}$ . If  $\Sigma_{A_1} \langle \sigma_1 \rangle \cap Shared(A_1, A_2) \neq \emptyset$ , then the transitions of  $\sigma_1$  and  $\sigma_2$  synchronize and produce autonomous exception-free runs between  $s_1^1 s_2^1$  and  $s_1^k s_2^l$ . For each  $\sigma$  from those runs, if  $\Upsilon_{A_{12}} \langle \sigma \rangle \cup \{Succ_{A_{12}}(s_1^k s_2^l, r_a)\} \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  if  $s_1 s_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $s_1 s_2 \notin \Upsilon_{A'_{12}}$ .

$\neq \emptyset$ , then all transitions labeled by shared actions of  $\sigma_1$  synchronize with their equivalents in  $A_2$  if they exist: either a deadlock state is hit and then  $s_1s_2 \notin \Upsilon_{A'_{12}}$ , or there is an autonomous exception-free run  $\sigma$  between  $s_1^1s_2^1$  and  $s_1^kt_2$  containing only actions in  $\Sigma_{A_1}\langle\sigma_1\rangle$  where  $t_2$  is reachable from  $s_2$  in  $A_2$ . In the latter case, if  $\Upsilon_{A_{12}}\langle\sigma\rangle \cup \{Succ_{A_{12}}(s_1^kt_2, r_a)\} \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  if  $s_1s_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $s_1s_2 \notin \Upsilon_{A'_{12}}$ . The same reasoning is adapted if  $a \in \Sigma_{A_2}^{lm}$ .

Finally, if  $\neg SemComp_a(A_1, A_2)$ ,  $s_1s_2$  is deadlock in  $A_{12}$  and removed in  $A'_{12}$ .

(2) If  $a \in \Sigma_{A_1}^m \setminus Shared(A_1, A_2)$ , we have  $R_{A_1}(a) = r_a \notin Shared(A_1, A_2)$ . The transition enabled from  $s_1^k$  labeled by  $r_a$  and that enabled from  $s_1$  labeled by  $a$  are interleaved in  $A_{12}$ . If  $\Sigma_{A_1}\langle\sigma_1\rangle \cap Shared(A_1, A_2) = \emptyset$ , the transitions of  $\sigma_1$  are interleaved and among the produced runs, we distinguish the autonomous exception-free run  $\sigma = s_1t_2[a]s_1^1t_2[a_1^1]...s_1^{k-1}t_2[a_1^{k-1}]s_1^kt_2$  in  $A_{12}$  where  $s_1^1t_2 = Succ(s_1t_2, a)$  and  $s_1t_2$  is reachable in  $A_{12}$ : if  $\Upsilon_{A_{12}}\langle\sigma\rangle \cup Succ_{A_{12}}(s_1^kt_2, r_a) \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  if  $s_1t_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $(s_1t_2, a, s_1^1t_2) \notin \delta_{A'_{12}}$ . If  $\Sigma_{A_1}\langle\sigma_1\rangle \cap Shared(A_1, A_2) \neq \emptyset$ , then all transitions labeled by shared actions of  $\sigma_1$  synchronize with their equivalents in  $A_2$  if they exist: either a deadlock state is hit and then all reachable states  $s_1t_2$  in  $A_{12}$  are removed in  $A'_{12}$ , or there is an autonomous exception-free run  $\sigma$  between all  $s_1t_2$  reachable in  $A_{12}$  and  $s_1^kt_2$  containing only actions in  $\{a\} \cup \Sigma_{A_1}\langle\sigma_1\rangle$ . In the latter case, if  $\Upsilon_{A_{12}}\langle\sigma\rangle \cup \{Succ_{A_{12}}(s_1^kt_2, r_a)\} \subseteq Cmp(A_1, A_2)$ , then  $\sigma$  remains in  $A'_{12}$  for all  $s_1t_2 \in Cmp(A_1, A_2)$ . Otherwise,  $\sigma$  is removed in  $A'_{12}$  and  $(s_1t_2, a, s_1^1t_2) \notin \delta_{A'_{12}}$ . The same reasoning is adapted if  $a \in \Sigma_{A_2}^{lm}$ . Consequently, from proofs (1) and (2), we can deduce that  $(B_1\|B_2).\mathcal{A}$  is well-formed.  $\square$

The following theorem is in the heart of incremental design of object-oriented component-based systems. It is a straightforward generalization of interface automata associativity [7] to behavioral contracts.

**Theorem 2.** *The composition operation  $\|$  between compatible behavioral contracts is commutative and associative.*

*Proof.* This proof is adapted from [8]. Let us consider three behavioral contracts  $B_1$ ,  $B_2$ , and  $B_3$  mutually composable and compatible. The proof of commutativity is trivial. It is also easy to check that  $((B_1\|B_2)\|B_3).\mathcal{I} = (B_1\|(B_2\|B_3)).\mathcal{I} = ((B_1\|B_3)\|B_2).\mathcal{I}$  and  $((B_1\|B_2)\|B_3).\mathcal{O} = (B_1\|(B_2\|B_3)).\mathcal{O} = ((B_1\|B_3)\|B_2).\mathcal{O}$ . The proof of associativity is mainly required at the level of interface automata. We denote  $B_1.\mathcal{A}$ ,  $B_2.\mathcal{A}$ , and  $B_3.\mathcal{A}$  resp. by  $A_1$ ,  $A_2$ , and  $A_3$ . We can obviously observe that the synchronization  $\otimes$  of interface automata is commutative and associative: we have  $(A_1 \otimes A_2) \otimes A_3 = A_1 \otimes (A_2 \otimes A_3) = (A_1 \otimes A_3) \otimes A_2 = A_1 \otimes A_2 \otimes A_3$  (denoted  $A_{123}$ ). We consider projections  $A_i \otimes A_j$  (denoted  $A_{ij}$ ) of  $A_{123}$  for  $i, j \in \{1, 2, 3\}$  and  $i \neq j$ . A state  $s_1s_2s_3$  is a deadlock state in  $A_{123}$  if  $s_is_j$  is a deadlock state in one of the projections  $A_{ij}$ . A state  $s_1s_2s_3$  is incompatible in  $A_{123}$  if there is a deadlock state  $d_1d_2d_3$  in  $A_{123}$  autonomously reachable from  $s_1s_2s_3$ . For  $l \geq 0$ , a state  $s_1s_2s_3$  is  $l$ -incompatible if there is a deadlock state  $d_1d_2d_3$  autonomously reachable from  $s_1s_2s_3$  by enabling at most  $l$  transitions.

It is sufficient to show that  $(B_1\|B_2\|B_3).\mathcal{A}$  (denoted  $A'_{123}$ ), for any insertion of parentheses, is the associative product  $A_{123}$  by removing incompatible states. We follow two steps: (1) we demonstrate that a state  $s_1s_2s_3$  is incompatible in  $A_{123}$  if there is a state  $s_is_j$  incompatible in one of its projection  $A_{ij}$ ; (2) we demonstrate that if there are transitions labeled by non-autonomous actions  $(\Sigma_{A_{ij}}^{lm} \cup (\Sigma_{A_{ij}}^l \cap \Sigma_{A_{ij}}^e))$  reachable autonomously from  $s_is_j$  and removed in  $(B_i\|B_j).\mathcal{A}$  (denoted  $A'_{ij}$ ), then in the product  $A_{123}$  without those transitions, there is always an autonomous run starting from  $s_1s_2s_3$  reaching a deadlock state.

(1) Given a state  $s_1s_2s_3$  in  $A_{123}$  and a projection  $s_is_j$  of  $s_1s_2s_3$   $k$ -incompatible in the product  $A_{ij}$ , we show that  $s_1s_2s_3$  is  $l'$ -incompatible with  $l' \leq l$  in  $A_{123}$ . Given  $\sigma$  the smallest autonomous run between  $s_is_j$  and a deadlock state  $d_id_j$  in  $A_{ij}$ . The proof is by induction: (base case 1) if  $s_is_j \in \text{Dead}(A_i, A_j)$ , then  $s_1s_2s_3$  is a deadlock state (0-incompatible in  $A_{123}$ ); (base case 2) if  $s_is_j \in \text{Comp}(A_i, A_j)$  and  $s_1s_2s_3 \in \text{Dead}(A_{ij}, A_k)$  for  $k \in \{1, 2, 3\} \setminus \{i, j\}$  (0-incompatible in  $A_{123}$ ); (step case) if the first transition of  $\sigma$  is labeled by an autonomous action, synchronized or interleaved in  $A_{123}$ , then the successor state  $t_1t_2t_3$  of  $s_1s_2s_3$  by enabling this action, is  $(l-1)$ -incompatible. The proof is iterated inductively until reaching a deadlock state  $d_1d_2d_3$  (one of the base cases).

(2) Given a state  $s = s_1s_2s_3$  incompatible in  $A_{123}$ , we assume that there are transitions labeled by non-autonomous actions  $a \in \Sigma_{A_{ij}}^{\text{Im}} \cup \Sigma_{A_{ij}}^{\text{Le}}$ , reachable from  $s_is_j$  in a sub-product  $A_{ij}$  and removed in  $A'_{ij}$  and  $A'_{123}$ . Only transitions  $(t, a, v)$  where  $a$  is hidden in  $A_{123}$  and their projections onto  $A_{ij}$  are transitions  $(t_it_j, a, v_iv_j)$  where  $a$  is non-autonomous and synchronized by its corresponding output action in  $A_k$  for  $k \in \{1, 2, 3\} \setminus \{i, j\}$ , can be removed by this way. Once  $(t, a, v)$  is removed from  $A_{123}$ , the input non-autonomous action is no longer enabled from  $t_it_j$  because interface automata are deterministic. Consequently, the state  $t \in \text{Dead}(A_{ij}, A_k)$ . Hence, after removing  $(t, a, v)$  from  $A_{123}$  there is always an autonomous run between  $s$  and a deadlock state, especially  $t$ .  $\square$

The compatibility check procedure of two behavioral contracts is similar to that described in [7] for interface automata, by considering the semantic layer of actions and the new definition of autonomous runs. The linear complexity of the proposed algorithm is extended by satisfiability decision problems of semantic compatibility conditions of shared input/output method actions. The original algorithm becomes a semi-algorithm on account of the various satisfiability problems which are either decidable (propositional logic) or not (arithmetic, etc).

## 5 Refinement

Refinement embodies with more details an abstract specification of a component in a more concrete one. It guarantees a safe substitutability of an abstract version of a component by a refined one. We propose a refinement approach for behavioral contracts, at protocol and semantic levels, suitable to the object-oriented context. We start by introducing refinement at the level of interface automata.

### 5.1 Expanding simulation

The original refinement relation of interface automata is *contravariant* [7] on output actions *i.e.*, it reverses sub-typing considered as inclusion of action sets: an abstract version of an interface automaton  $A$  issues the same or more outputs, and accepts the same or fewer inputs than its refined version  $A'$ . It is based on an alternating simulation relation [4]: each output event of  $A'$  is simulated by at least the same event in  $A$ , and each input event of  $A$  is simulated exactly by the same in  $A'$ . At the protocol level in OOCBD, our approach of refinement ensures that a refined specification of a component may (i) provide more methods than the abstract one, and (ii) contain more details about common provided methods with the abstract one, which are output and local method calls encapsulated in their bodies. These requirements lead us to consider refinement relation to be *covariant* on both input and output actions *i.e.*, it preserves sub-typing for both kinds of action sets:  $A'$  refines  $A$  if  $A$  accepts (resp. issues) fewer inputs

(resp. outputs) than  $A'$ . We base our approach on the so called expanding simulation: each input, output, or local event of  $A$  is simulated in  $A'$  by the same one followed or preceded by other events.

To formalize this relation, we define the *closure* set  $Clos_A(s, \Sigma)$  of  $s \in \Upsilon_A$  under actions in  $\Sigma \subseteq \Sigma_A$  by the largest set  $\Upsilon \subseteq \Upsilon_A$  such that  $s \in \Upsilon$  and if  $t \in \Upsilon$ ,  $v = Succ_A(t, a)$ , and  $a \in \Sigma$ , then  $v \in \Upsilon$  i.e.,  $Clos_A(s, \Sigma)$  contains states reachable from the state  $s$  by enabling actions of  $\Sigma$ .

**Definition 10.** *Given two interface automata  $A$  and  $A'$ , a binary relation  $\succsim \subseteq \Upsilon_A \times \Upsilon_{A'}$  is an expanding simulation from  $A$  to  $A'$  iff for all  $ss' \in \Upsilon_A \times \Upsilon_{A'}$  and  $a \in \Sigma_A(s)$  where  $s \succsim s'$  and  $t = Succ_A(s, a)$ , the following conditions hold:*

- (1) *if  $a \in \Sigma_A^{Om}(s) \cup \Sigma_A^{Ir}(s) \cup \Sigma_A^{Ie}(s)$ , then  $a \in \Sigma_{A'}(s')$  and  $t \succsim t'$  for  $t' = Succ_{A'}(s', a)$ ;*
- (2) *if  $a \in \Sigma_A^{Im}(s) \cup \Sigma_A^{Hm}(s)$ , then  $a \in \Sigma_{A'}(s')$ , and there is a set  $\Sigma \subseteq ((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$  and a state  $t' \in Clos_{A'}(Succ_A(s', a), \Sigma)$  such that  $t \succsim t'$ ;*
- (3) *if  $a \in \Sigma_A^{Or}(s) \cup \Sigma_A^{Hr}(s)$ , then there is a state  $v' \in Clos_{A'}(s', \Sigma)$  such that  $a \in \Sigma_{A'}(v')$ ,  $\Sigma = ((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$ , and  $t \succsim t'$  for  $t' = Succ_{A'}(v', a)$ ;*
- (4) *if  $a \in \Sigma_A^{Oe}(s) \cup \Sigma_A^{He}(s)$ , then there is a state  $v' \in Clos_{A'}(s', \Sigma)$  such that  $a \in \Sigma_{A'}(v')$ ,  $\Sigma = ((\Sigma_{A'}^{aut} \setminus (\Sigma_{A'}^{Oe} \cup \Sigma_{A'}^{Or})) \cup \Sigma_{A'}^{Ie}) \setminus \Sigma_A$ , and  $t \succsim t'$  for  $t' = Succ_{A'}(v', a)$ .*

Our expanding simulation relation pinpoints where refinement details are added in the abstract version of an interface automaton. Condition (1) of Definition 10 states that every transition labeled by an output method action, or an input return or exception action must be matched by a transition labeled by the same action in  $A'$ . Method calls sent to the environment, the reception of their return values, and catching their thrown exceptions, cannot be refined.

Condition (2) states that every transition labeled by an input or hidden method action in  $A$  is matched in  $A'$  by a transition labeled by the same action followed by zero or more transitions labeled by a “subset” of new autonomous non-exception actions in  $((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$ . A provided public method in the abstraction of a component can be refined by adding to its body new private or public method calls. In addition, since providing private methods is not specified by actions in interface automata (cf. Section 3), our simulation relation allows adding refinement details about private methods after their calls.

Condition (3) states that every transition labeled by an output or hidden return action  $a$  in  $A$  is matched in  $A'$  by zero or more transitions labeled by new autonomous non-exception actions in  $((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$  followed by a transition labeled by  $a$ . The return event of a private or public provided method in the abstraction is computed based on the return values of new calls of private or public methods added as refinement details.

Condition (4) states that every transition labeled by an output or hidden exception action  $a$  in  $A$  is matched in  $A'$  by zero or more transitions labeled either by new autonomous and hidden exception actions in  $(\Sigma_{A'}^{aut} \setminus (\Sigma_{A'}^{Oe} \cup \Sigma_{A'}^{Or})) \setminus \Sigma_A$ , or by new input exception actions in  $\Sigma_{A'}^{Ie} \setminus \Sigma_A$ , followed by a transition labeled by  $a$ . The exception events of a provided private or public method in the abstraction is the propagation of catching exception events of new calls of private or public methods added as refinement details.

We infer from conditions of Definition 10 that extra new input method actions are not considered as refinement details by the expanding simulation relation, which obviously makes sense. By cons, it allows the extension of interface automata by adding protocols related to additional methods provided by a component extended interface. They can be enabled for example separately from the initial state. We define the refinement relation as follows.

**Definition 11.**  $A'$  refines  $A$  ( $A \succeq A'$ ) iff

- (1)  $\Sigma_A^I \subseteq \Sigma_{A'}^I$ ,  $\Sigma_A^O \subseteq \Sigma_{A'}^O$ ,  $\Sigma_A^H \subseteq \Sigma_{A'}^H$ , and
- (2) there is an expanding simulation  $\succeq$  from  $A$  to  $A'$  such that  $\iota_A \succeq \iota_{A'}$ .

A trivial consequence of condition (1) of Definition 11 is covariance from  $A$  to  $A'$  on method, return, and exception actions:  $\Sigma_A^m \subseteq \Sigma_{A'}^m$ ,  $\Sigma_A^r \subseteq \Sigma_{A'}^r$ , and  $\Sigma_A^e \subseteq \Sigma_{A'}^e$ . Condition (2) requires the existence of an expanding simulation from  $A$  to  $A'$  relating their initial states  $\iota_A$  and  $\iota_{A'}$  and recursively propagated to their successor states.

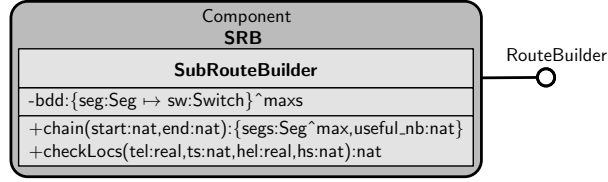


Figure 7: Extended version of SubRouteBuilder.

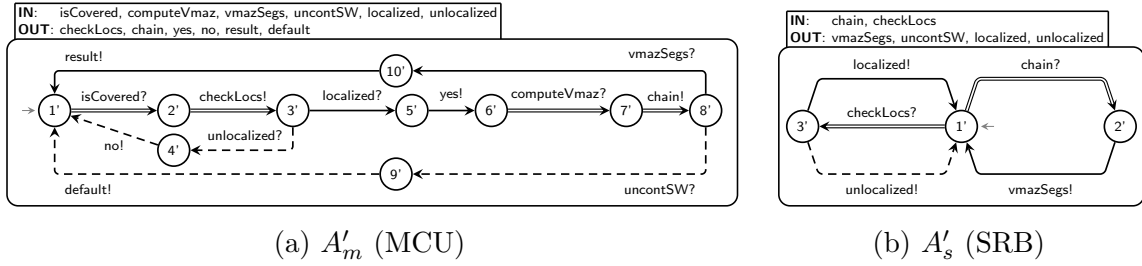


Figure 8: Refined interface automata of MCU and SRB.

**Example 5.** After receiving a train coverage request, MCU asks SRB to check if  $tel$  and  $hel$  are really on segments  $ts$  and  $hs$  respectively by calling the method  $checkLocs$ , presented in Figure 8(left), as a new service of the class `SubRouteBuilder` and the interface `RouteBuilder`. If true, SRB responds by sending the status ( $localized$ ), and MCU in turn, responds OBD, via DEU, by returning  $yes$  if the train is completely (or partially) included in its coverage area. Otherwise, SRB throws the exception  $unlocalized$  to MCU, which in turn, propagates it to DEU by throwing the exception  $no$ . In  $A'_m$  shown in Figure 8(a), the method call  $checkLocs!$  is encapsulated in runs describing the body of  $isCovered$  provided by MCU. Providing the public method  $checkLocs$  is equally depicted in the interface automaton  $A'_s$  shown in Figure 8(b) by a new input method action enabled separately from  $\iota_{A'_s} = 1'$ .  $A'_m$  and  $A'_s$  resp. refine  $A_m$  and  $A_s$  (shown resp. in Figure 3(c) and Figure 5): condition (1) of Definition 11 is met by  $A'_m$  and  $A_m$ , as well by  $A'_s$  and  $A_s$ , and there are two expanding simulations  $\succeq_m = \{11', 23', 36', 47', 58', 69', 7(10')\}$  from  $A_m$  to  $A'_m$  with  $\iota_{A_m} \succeq_m \iota_{A'_m}$  and  $\succeq_s = \{11', 22'\}$  from  $A_s$  to  $A'_s$  with  $\iota_{A_s} \succeq_s \iota_{A'_s}$ .

## 5.2 Semantic substitutability

The semantic substitutability of method actions between an abstract and a concrete versions of a component behavioral contract is based on *behavioral sub-typing principles* introduced in [5, 12]: in the refined specification, a common provided method must have a weaker precondition, a stronger termination postcondition, and does not introduce exceptions by supplying a stronger exception condition, than the abstract one. Inversely, a common method call must have, in

refinement, a stronger precondition and a weaker postcondition than abstraction. Given two behavioral contracts  $B$  and  $B'$ , we denote  $B.\mathcal{A}$  by  $A$  and  $B'.\mathcal{A}$  by  $A'$ .

**Definition 12.** *Given an action  $a \in \Sigma_A^{\text{Im}}$ ,  $B'.\mathcal{I}(a) = (P'_a, S'_a, Q'_a, E'_a)$  substitutes  $B.\mathcal{I}(a) = (P_a, S_a, Q_a, E_a)$  i.e.,  $\text{SemSub}_a(B, B')$ , iff for all  $(i, f, o) \in \Psi_A^i(a) \times \Lambda_A(a) \times \Psi_A^o(a)$ , the following conditions hold:*

- (1)  $P_a[i] \Rightarrow P'_a[i], Q_a[i, f, o] \Leftarrow Q'_a[i, f, o], E_a[i, f, o] \Leftarrow E'_a[i, f, o]$ , and
- (2)  $P_a[i] \wedge S'_a[i, f, o] \Rightarrow S_a[i, f, o]$ .

*Given  $b \in \Sigma_A^{\text{Om}}$ ,  $B'.\mathcal{O}(b) = (P'_b, Q'_b)$  substitutes  $B.\mathcal{O}(b) = (P_b, Q_b)$  i.e.,  $\text{SemSub}_b(B, B')$  iff for all  $(i, o) \in \Psi_A^i(b) \times \Psi_A^o(b)$ , the following condition holds:*

- (3)  $P_b[i] \Leftarrow P'_b[i]$  and  $Q_b[i, o] \Rightarrow Q'_b[i, o]$ .

The following property is evident based on definitions 5 and 12. Correctness, exception, and termination conditions preservation is what we expect for a correct refinement at the semantic level of provided methods: if the refined semantics of a provided method satisfies condition (2) of Definition 12, then any property holding for a specification  $S$  under the precondition in the abstract method semantics, holds also for the refined specification  $S'$  under the same precondition, and thus  $S'$  may be used instead of  $S$  [13, 3].

**Property 1.** *Given  $a \in \Sigma_A^{\text{Im}}$  where  $\text{SemSub}_a(B, B')$ , for all  $(i, f, o) \in \Psi_A^i(a) \times \Lambda_A(a) \times \Psi_A^o(a)$ ,*

- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow Q_a[i, f, o]$ ,
- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow E_a[i, f, o]$ , or
- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow Q_a[i, f, o] \wedge \neg E_a[i, f, o]$ .

*resp. iff  $a$  is correct, throws exceptions, or terminates with respect to  $B.\mathcal{I}(a)$ .*

Property 2 says that the semantic compatibility validity of shared observable method actions, between behavioral contracts of a component and its environment, is preserved by the semantic substitutability. The property is obvious based on Definition 6 and conditions (1) and (3) of Definition 12. Given a behavioral contract  $E$ , we set  $E.\mathcal{A} = A_E$ .

**Property 2.** *Given an action  $a \in \text{Shared}(A, A_E) \cap \Sigma_A^{\text{m}}$ , for all  $(i, o) \in \Psi_A^i(a) \times \Psi_A^o(a)$ , if  $\text{SemSub}_a(B, B')$ , then  $\text{SemComp}_a(B, E) = \text{SemComp}_a(B', E)$ .*

Finally, we can define refinement of behavioral contracts based on refinement of interface automata and the semantic substitutability of observable method actions.

**Definition 13.**  $B'$  refines  $B$  ( $B \sqsupseteq B'$ ) iff  $A \succeq A'$  and for all  $a \in \Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Om}}$ ,  $\text{SemSub}_a(B, B')$ .

### 5.3 Refinement properties

In this subsection, we present the properties and requirements under which our refinement approach allows independent implementability of components using their behavioral contracts.

### Reflexivity and transitivity

Lemma 3 states that the expanding simulation between interface automata is a transitive relation. This result is necessary to prove that the refinement relation  $\sqsubseteq$  is a preorder (Theorem 4) *i.e.*, a behavioral contract can be gradually refined in several steps while remaining consistent with its abstract specification.

**Lemma 3.** *given three interface automata  $A$ ,  $A'$ , and  $A''$ , and two expanding simulations  $\succsim' \subseteq \Upsilon_A \times \Upsilon_{A'}$  and  $\succsim'' \subseteq \Upsilon_{A'} \times \Upsilon_{A''}$ , then the composite relation  $\succsim'' \circ \succsim' \subseteq \Upsilon_A \times \Upsilon_{A''}$  is an expanding simulation.*

*Proof.* We set some notations. Given an interface automaton  $M$ , a state  $s \in \Upsilon_M$ , and a set of actions  $\Sigma \subseteq \Sigma_M$ , we define recursively  $Clos_M^k(s, \Sigma)$  by:  $Clos_M^0(s, \Sigma) = \{s\}$  and  $Clos_M^k(s, \Sigma) = Clos_M^{k-1}(s, \Sigma) \cup \{t = Succ_M(s, a) \mid a \in \Sigma \wedge s \in Clos_M^{k-1}(s, \Sigma)\}$  for  $k > 0$ . We prove that  $\succsim'' \circ \succsim' = \{ss'' \in \Upsilon_A \times \Upsilon_{A''} \mid (\exists s' \in \Upsilon_{A'} \mid s \succsim' s' \succsim'' s'')\}$  is an expanding simulation. For all  $s \in \Upsilon_A$ , based on Definition 10, we state the following properties for all  $a \in \Sigma_A(s)$  and  $t = Succ_A(s, a)$ :

- (1) if  $a \in \Sigma_A^{Om}(s) \cup \Sigma_A^{Ir}(s) \cup \Sigma_A^{Ie}(s)$ ,  $a \in \Sigma_{A'}(s')$ ,  $a \in \Sigma_{A''}(s'')$ , and  $t \succsim' t' \succsim'' t''$  for  $t' = Succ_{A'}(s', a)$  and  $t'' = Succ_{A''}(s'', a)$ , that is  $t \succsim'' \circ \succsim' t''$ ;
- (2) if  $a \in \Sigma_A^{Im}(s) \cup \Sigma_A^{Hm}(s)$ , then  $a \in \Sigma_{A'}(s')$ , and there is a set  $\Sigma' \subseteq ((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$  and a state  $t' \in Clos_{A'}(Succ_{A'}(s', a), \Sigma')$  such that  $t \succsim' t'$ . We have to prove that  $a \in \Sigma_{A''}(s'')$ , and there is a set  $\Sigma \subseteq ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_A$  and a state  $t'' \in Clos_{A''}(Succ_{A''}(s'', a), \Sigma)$  such that  $t \succsim' t' \succsim'' t''$ ? We consider  $Clos_{A'}^k(Succ_{A'}(s', a), \Sigma') \subseteq Clos_{A'}(Succ_{A'}(s', a), \Sigma')$  where  $k$  is the first natural such that  $t' \in Clos_{A'}^k(Succ_{A'}(s', a), \Sigma')$ . We define the states  $s'_i \in Clos_{A'}^k(Succ_{A'}(s', a), \Sigma') \cup \{s'\}$  inductively on  $0 \leq i \leq k+1$ :  $s'_0 = s$ ,  $s'_1 = Succ_{A'}(s'_0, a)$ ,  $s'_i \in Clos_{A'}^{i-1}(s'_1, \Sigma') \setminus Clos_{A'}^{i-2}(s'_1, \Sigma')$  for  $i > 1$ . We define  $s''_i$  and  $\Upsilon''_i$  inductively on  $i$ :
  - (2.1)  $i = 0$ :  $s''_0 = s''$  (we recall that  $s'_0 \succsim'' s''_0$ );  $\Upsilon''_0 = \{s''_0\}$ ;
  - (2.2)  $i = 1$ :  $a \in \Sigma_{A''}(s''_0)$ , there is a set  $\Sigma''_0 \subseteq ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_{A'}$  and a state  $s''_1 \in Clos_{A''}(Succ_{A''}(s''_0, a), \Sigma''_0)$  such that  $s'_1 \succsim'' s''_1$ ;  $\Upsilon''_1 = \Upsilon''_0 \cup Clos_{A''}(Succ_{A''}(s''_0, a), \Sigma''_0)$ ;
  - (2.3)  $i > 1$ : for all  $b_{i-1} \in \Sigma_{A'}(s'_{i-1}) \cap \Sigma'$  and  $s'_i = Succ_{A'}(s'_{i-1}, b_{i-1})$ , we have three cases:
    - (2.3.1) if  $b_{i-1} \in \Sigma_{A'}^{Om}(s'_{i-1}) \cup \Sigma_{A'}^{Ir}(s'_{i-1})$ , then  $b_{i-1} \in \Sigma_{A''}(s''_{i-1})$  and  $s'_i \succsim'' s''_i$  where the state  $s''_i = Succ_{A''}(s''_{i-1}, b_{i-1})$ ;  $\Upsilon''_i = \Upsilon''_{i-1} \cup \{s''_i = Succ_{A''}(s''_{i-1}, b_{i-1}) \mid b_{i-1} \in (\Sigma_{A'}^{Om}(s'_{i-1}) \cup \Sigma_{A'}^{Ir}(s'_{i-1})) \cap \Sigma'\}$ ;
    - (2.3.2) if  $b_{i-1} \in \Sigma_{A'}^{Hm}(s'_{i-1})$ ,  $b_{i-1} \in \Sigma_{A''}(s''_{i-1})$ , and there is  $\Sigma''_{i-1} \subseteq ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_{A'}$  and  $s''_i \in Clos_{A''}(Succ_{A''}(s''_{i-1}, b_{i-1}), \Sigma''_{i-1})$  such that  $s'_i \succsim'' s''_i$ ;  $\Upsilon''_i = \Upsilon''_{i-1} \cup (\bigcup_{\phi} Clos_{A''}(Succ_{A''}(s''_{i-1}, b_{i-1}), \Sigma''_{i-1}))$  where  $\phi \equiv b_{i-1} \in \Sigma_{A'}^{Hm}(s'_{i-1}) \cap \Sigma'$ ;
    - (2.3.3) if  $b_{i-1} \in \Sigma_{A'}^{Hr}(s'_{i-1})$ , there is a state  $v''_{i-1} \in Clos_{A''}(s''_{i-1}, \Sigma''_{i-1})$  such that  $b_{i-1} \in \Sigma_{A''}(v''_{i-1})$ ,  $\Sigma''_{i-1} = ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_{A'}$ , and  $s'_i \succsim'' s''_i$  for the state  $s''_i = Succ_{A''}(v''_{i-1}, b_{i-1})$ ;  $\Upsilon''_i = \Upsilon''_{i-1} \cup (\bigcup_{\phi} Clos_{A''}(s''_{i-1}, \Sigma''_{i-1})) \cup \{Succ_{A''}(v''_{i-1}, b_{i-1}) \mid (\forall v''_{i-1} \mid \phi)\}$  where  $\phi \equiv b_{i-1} \in \Sigma_{A'}^{Hr}(s'_{i-1}) \cap \Sigma'$ .

We deduce from (2.1) and (2.2) that  $a \in \Sigma_{A''}(s'')$ . Given  $s'_k \in Clos_{A'}^k(Succ_{A'}(s', a), \Sigma')$  where  $s'_k = t'$ , according to (2.3), there is a set  $\Upsilon''_{k+1} \subseteq Clos_{A''}(Succ_{A''}(s''_k, a), \Sigma)$  where  $\Sigma \subseteq ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_A$  and  $t'' \in \Upsilon''_{k+1}$  such that  $t \succsim' t' \succsim'' t''$ , that is  $t \succsim'' \circ \succsim' t''$ .

- (3) if  $a \in \Sigma_A^{Or}(s) \cup \Sigma_A^{Hr}(s)$ , there is  $v' \in Clos_{A'}(s', \Sigma')$  such that  $\Sigma' = ((\Sigma_{A'}^{aut} \setminus \Sigma_{A'}^{Or}) \setminus \Sigma_{A'}^e) \setminus \Sigma_A$ ,  $a \in \Sigma_{A'}(v')$ , and  $t \succsim' t'$  for  $t' = Succ_{A'}(v', a)$ . We have to prove that there is a state  $v'' \in Clos_{A''}(s'', \Sigma)$  such that  $\Sigma = ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_A$ ,  $a \in \Sigma_{A''}(v'')$ , and  $t \succsim' t' \succsim'' t''$  for  $t'' = Succ_{A''}(v'', a)$ ? We consider  $Clos_{A'}^k(s', \Sigma') \subseteq Clos_{A'}(s', \Sigma')$  where  $k$  is the first



natural such that  $v' \in Clos_{A'}^k(s', \Sigma')$ . We define the states  $s'_i \in Clos_{A'}^k(s', \Sigma')$  inductively on  $0 \leq i \leq k$ :  $s'_0 = s'$  and  $s'_i \in Clos_{A'}^i(s', \Sigma') \setminus Clos_{A'}^{i-1}(s', \Sigma')$  for  $i > 0$ . For all  $b_i \in \Sigma_{A'}(s'_i) \cap \Sigma'$  and  $s'_{i+1} = Succ_{A'}(s'_i, b_i)$ , we define equally  $s''_i$  and  $\Upsilon''_i$  inductively on  $i$ :

- (3.1)  $i = 0$ :  $s''_0 = s''$  ( $s'_0 \succ'' s''_0$ );  $\Upsilon''_0 = \{s''_0\}$ ;
- (3.2)  $i > 0$ : for all  $b_{i-1} \in \Sigma_{A'}(s'_{i-1}) \cap \Sigma'$  and  $s'_i = Succ_{A'}(s'_{i-1}, b_{i-1})$ , we have three cases:
  - (3.2.1) if  $b_{i-1} \in \Sigma_{A'}^{Om}(s'_{i-1}) \cup \Sigma_{A'}^{Ir}(s'_{i-1})$ , then the definition goes as in (2.3.1);
  - (3.2.2) if  $b_{i-1} \in \Sigma_{A'}^{Hm}(s'_{i-1})$ , then the definition goes as in (2.3.2);
  - (3.2.3) if  $b_{i-1} \in \Sigma_{A'}^{Hr}(s'_{i-1})$ , then the definition goes as in (2.3.3).

Given  $s'_k \in Clos_{A'}^k(s', \Sigma')$  where  $s'_k = v'$ , and  $s''_k \in \Upsilon''_k$  where  $s'_k \succ'' s''_k$ , according to (3.2), there is  $v''_k \in \Upsilon''_k \subseteq Clos_{A''}(s'', ((\Sigma_{A''}^{aut} \setminus \Sigma_{A''}^{Or}) \setminus \Sigma_{A''}^e) \setminus \Sigma_A)$  such that  $a \in \Sigma_{A''}(v''_k)$ , and  $t \succ' t' \succ'' t''$  for  $t'' = Succ_{A''}(v''_k, a)$ , that is  $t \succ'' \circ \succ' t''$ .

- (4) if  $a \in \Sigma_A^{Oe}(s) \cup \Sigma_A^{He}(s)$ , there is  $v' \in Clos_{A'}(s', \Sigma')$  such that  $\Sigma' = ((\Sigma_{A'}^{aut} \setminus (\Sigma_{A'}^{Oe} \cup \Sigma_{A'}^{Or})) \cup \Sigma_{A'}^{Ie}) \setminus \Sigma_A$ ,  $a \in \Sigma_{A'}(v')$ , and  $t \succ' t'$  for  $t' = Succ_{A'}(v', a)$ . We have to prove that there is a state  $v'' \in Clos_{A''}(s'', \Sigma)$  such that  $\Sigma = ((\Sigma_{A''}^{aut} \setminus (\Sigma_{A''}^{Oe} \cup \Sigma_{A''}^{Or})) \cup \Sigma_{A''}^{Ie}) \setminus \Sigma_A$ ,  $a \in \Sigma_{A''}(v'')$ , and  $t \succ' t' \succ'' t''$  for  $t'' = Succ_{A''}(v'', a)$ ? We consider  $Clos_{A'}^k(s', \Sigma') \subseteq Clos_{A'}(s', \Sigma')$  where  $k$  is the first natural such that  $v' \in Clos_{A'}^k(s', \Sigma')$ . We define the states  $s'_i \in Clos_{A'}^k(s', \Sigma')$  inductively on  $0 \leq i \leq k$ :  $s'_0 = s'$  and  $s'_i \in Clos_{A'}^i(s', \Sigma') \setminus Clos_{A'}^{i-1}(s', \Sigma')$  for  $i > 0$ . For all  $b_i \in \Sigma_{A'}(s'_i) \cap \Sigma'$  and  $s'_{i+1} = Succ_{A'}(s'_i, b_i)$ , we define equally  $s''_i$  and  $\Upsilon''_i$  inductively on  $i$ :

- (4.1)  $i = 0$ :  $s''_0 = s''$  ( $s'_0 \succ'' s''_0$ );  $\Upsilon''_0 = \{s''_0\}$ ;
- (4.2)  $i > 0$ : for all  $b_{i-1} \in \Sigma_{A'}(s'_{i-1}) \cap \Sigma'$  and  $s'_i = Succ_{A'}(s'_{i-1}, b_{i-1})$ , we have four cases:
  - (4.2.1) if  $b_{i-1} \in \Sigma_{A'}^{Om}(s'_{i-1}) \cup \Sigma_{A'}^{Ir}(s'_{i-1}) \cup \Sigma_{A'}^{Ie}(s'_{i-1})$ , then the definition goes as in (2.3.1);
  - (4.2.2) if  $b_{i-1} \in \Sigma_{A'}^{Hm}(s'_{i-1})$ , then the definition goes as in (2.3.2);
  - (4.2.3) if  $b_{i-1} \in \Sigma_{A'}^{Hr}(s'_{i-1})$ , then the definition goes as in (2.3.3);
  - (4.2.4) if  $b_{i-1} \in \Sigma_{A'}^{He}(s'_{i-1})$ , then there is a state  $v''_{i-1} \in Clos_{A''}(s''_{i-1}, \Sigma''_{i-1})$  such that  $b_{i-1} \in \Sigma_{A''}(v''_{i-1})$ ,  $\Sigma''_{i-1} = ((\Sigma_{A''}^{aut} \setminus (\Sigma_{A''}^{Oe} \cup \Sigma_{A''}^{Or})) \cup \Sigma_{A''}^{Ie}) \setminus \Sigma_{A'}$ , and  $s'_i \succ'' s''_i$  for  $s''_i = Succ_{A''}(v''_{i-1}, b_{i-1})$ ;  $\Upsilon''_i = \Upsilon''_{i-1} \cup (\bigcup_{\phi} Clos_{A''}(s''_{i-1}, \Sigma''_{i-1})) \cup \{Succ_{A''}(v''_{i-1}, b_{i-1}) \mid (\forall v''_{i-1} \mid \phi)\}$  where  $\phi \equiv b_{i-1} \in \Sigma_{A'}^{He}(s'_{i-1}) \cap \Sigma'$ .

Given  $s'_k \in Clos_{A'}^k(s', \Sigma')$  where  $s'_k = v'$ , and  $s''_k \in \Upsilon''_k$  where  $s'_k \succ'' s''_k$ , according to (4.2), there is  $v''_k \in \Upsilon''_k \subseteq Clos_{A''}(s'', ((\Sigma_{A''}^{aut} \setminus (\Sigma_{A''}^{Oe} \cup \Sigma_{A''}^{Or})) \cup \Sigma_{A''}^{Ie}) \setminus \Sigma_A)$  such that  $a \in \Sigma_{A''}(v''_k)$ , and  $t \succ' t' \succ'' t''$  for  $t'' = Succ_{A''}(v''_k, a)$ , that is  $t \succ'' \circ \succ' t''$ .

From (1), (2), (3), and (4), we deduce that  $\succ'' \circ \succ' \subseteq \Upsilon_A \times \Upsilon_{A''}$  is an expanding simulation.  $\square$

**Theorem 4.** *The refinement relation  $\sqsupseteq$  between behavioral contracts is a preorder i.e., reflexive and transitive.*

*Proof.* Given three behavioral contracts  $B$ ,  $B'$ , and  $B''$  where  $B.A = A$ ,  $B'.A = A'$ , and  $B''.A = A''$ . We have to prove that  $B \sqsupseteq B$  (reflexivity) and if  $B \sqsupseteq B'$  and  $B' \sqsupseteq B''$ , then  $B \sqsupseteq B''$  (transitivity)? For reflexivity, it is trivial that  $A \succeq A$  and for all  $a \in \Sigma_A^m$ ,  $SemSub_a(B, B)$ . For transitivity, we have to prove that (1) if  $A \succeq A'$  and  $A' \succeq A''$ , then  $A \succeq A''$ , and (2) for all  $a \in \Sigma_A^m$ , if  $SemSub_a(B, B')$  and  $SemSub_a(B', B'')$ , then  $SemSub_a(B, B'')$ ?

(1) Based on Definition 11 and the assumptions  $A \succeq A'$  and  $A' \succeq A''$ , we can deduce that  $\Sigma_A^m \subseteq \Sigma_{A''}^m$ ,  $\Sigma_A^r \subseteq \Sigma_{A''}^r$ ,  $\Sigma_A^e \subseteq \Sigma_{A''}^e$ ,  $\Sigma_A^l \subseteq \Sigma_{A''}^l$ ,  $\Sigma_A^o \subseteq \Sigma_{A''}^o$ , and  $\Sigma_A^h \subseteq \Sigma_{A''}^h$ . It remains to prove that there is an expanding simulation  $\succ \subseteq \Upsilon_A \times \Upsilon_{A''}$  such that  $\iota_A \succ \iota_{A''}$ . We have, as assumptions, two expanding simulations  $\succ' \subseteq \Upsilon_A \times \Upsilon_{A'}$  and  $\succ'' \subseteq \Upsilon_{A'} \times \Upsilon_{A''}$  such that  $\iota_A \succ' \iota_{A'} \succ'' \iota_{A''}$ . We choose the composite relation  $\succ'' \circ \succ' \subseteq \Upsilon_A \times \Upsilon_{A''}$ . From Lemma 3,  $\succ'' \circ \succ'$  is an expanding simulation such that  $\iota_A \succ'' \circ \succ' \iota_{A''}$ .

(2) Based on Definition 12 and the assumptions  $SemSub_a(B, B')$  and  $SemSub_a(B', B'')$  for all  $a \in \Sigma_A^m$ , we can deduce that if  $a \in \Sigma_A^{Im}$ , then for all  $(i, f, o) \in \Psi_A^i(a) \times \Lambda_A(a) \times \Psi_A^o(a)$ ,  $P_a[i] \Rightarrow P'_a[i]$  (from  $P_a[i] \Rightarrow P'_a[i] \Rightarrow P''_a[i]$ ),  $Q_a[i, f, o] \Leftarrow Q'_a[i, f, o]$  (from  $Q_a[i, f, o] \Leftarrow Q'_a[i, f, o] \Leftarrow Q''_a[i, f, o]$ ),  $E_a[i, f, o] \Leftarrow E'_a[i, f, o]$  (from  $E_a[i, f, o] \Leftarrow E'_a[i, f, o] \Leftarrow E''_a[i, f, o]$ ), and  $P_a[i] \wedge S''_a[i, f, o] \Rightarrow S_a[i, f, o]$  (from  $P_a[i] \wedge S'_a[i, f, o] \Rightarrow S_a[i, f, o]$ ,  $P'_a[i] \wedge S''_a[i, f, o] \Rightarrow S'_a[i, f, o]$ , and  $P_a[i] \Rightarrow P'_a[i]$ ) where  $B.\mathcal{I}(a) = (P_a, B_a, Q_a, E_a)$ ,  $B'.\mathcal{I}(a) = (P'_a, B'_a, Q'_a, E'_a)$ , and  $B''.\mathcal{I}(a) = (P''_a, B''_a, Q''_a, E''_a)$ . Else if  $a \in \Sigma_A^{Om}$ , then for all  $(i, o) \in \Psi_A^i(a) \times \Psi_A^o(a)$ ,  $P_a[i] \Leftarrow P'_a[i]$  (from  $P_a[i] \Leftarrow P'_a[i] \Leftarrow P''_a[i]$ ) and  $Q_a[i, o] \Rightarrow Q'_a[i, o]$  (from  $Q_a[i, o] \Rightarrow Q'_a[i, o] \Rightarrow Q''_a[i, o]$ ) where  $B.\mathcal{O}(a) = (P_a, Q_a)$ ,  $B'.\mathcal{O}(a) = (P'_a, Q'_a)$ ,  $B''.\mathcal{O}(a) = (P''_a, Q''_a)$ .  $\square$

## Independent implementability

Refinement is expected to allow independent implementability of components: compatible behavioral contracts can be refined separately, while still maintaining compatibility. It lets industries unrestricted to outsource the implementation of components by different suppliers, after the refinement process, even if they do not communicate [8].

Our refinement approach guarantees the consistency between two behavioral contracts  $B$  and  $B'$  where  $B \sqsupseteq B'$  if they are considered “isolated” from their environment. However, it is highly generic, and does not prevent the introduction of poorly designed behaviors in their interface automata. Since refinement may issues new outputs, the designer should “safely” define it to preserve compatibility with the environment within the abstraction is incorporated without altering their communication scenarios. For example, according to Definition 2 and conditions (2) and (3) of Definition 10, the proposed expanding simulation relation preserves well-formedness in refinement only for method actions events common with abstraction. By cons, it does not guarantee that new method actions events are followed necessarily by their return events. In general, the higher the refinement design respects environment requirements and well-formedness, the safer it is considered to be.

We provide the weakest conditions under which the refinement of a behavioral contract is considered to be safe with respect to its environment. To formalize these requirements, let us consider three behavioral contracts  $B_1$ ,  $B'_1$  and  $B_2$  where  $B_1$  and  $B_2$  are composable and compatible,  $B_1 \sqsupseteq B'_1$ , and  $B'_1$  is composable with  $B_2$ . Let  $B_1.\mathcal{A} = A_1$ ,  $B'_1.\mathcal{A} = A'_1$ ,  $B_2.\mathcal{A} = A_2$ ,  $(B_1|B_2).\mathcal{A} = A_{12}$ , and  $(B'_1|B_2).\mathcal{A} = A'_{12}$ , we set  $EnabRiseDead(A_1, A_2) = \{a \in (\Sigma_{A_{12}}^{Im} \cup \Sigma_{A_{12}}^{Le}) \cap \Sigma_{A_{12}}\langle\sigma\rangle \mid \sigma \in \Theta_{A_{12}}(d_1d_2), d_1d_2 \in Dead(A_1, A_2)\}$ : the set of non-autonomous actions enabled by runs  $\sigma \in \Theta_{A_{12}}(d_1d_2)$  for all  $d_1d_2 \in Dead(A_1, A_2)$ . Since  $B_1$  and  $B_2$  are compatible,  $EnabRiseDead(A_1, A_2) \neq \emptyset$  if  $Dead(A_1, A_2) \neq \emptyset$ . Given  $\succsim$  an expanding simulation from  $A_1$  to  $A'_1$  such that  $\iota_{A_1} \succsim \iota_{A'_1}$ , we state the following definition.

**Definition 14.**  $B'_1$  is a safe refinement of  $B_1$  with respect to  $B_2$ , denoted  $B_1 \sqsupseteq_{B_2}^s B'_1$ , if the following conditions hold on  $A_1$ ,  $A'_1$ , and  $A_2$ :

- (1) for all  $d'_1d_2 \in Dead(A'_1, A_2)$ , there is a state  $d_1d_2 \in Dead(A_1, A_2)$  such that  $d_1 \succsim d'_1$  or  $d'_1 \in Clos_{A'_1}(c'_1, \Sigma_{A'_1} \setminus \Sigma_{A_1})$  and  $d_1 \succsim c'_1$ ;
- (2)  $Shared(A'_1, A_2) \cap EnabRiseDead(A_1, A_2) = \emptyset$ .

Condition (1) says that  $A'_{12}$  does not introduce new deadlocks compared to  $A_{12}$  by guaranteeing that all states in  $Dead(A'_1, A_2)$  are simulated by states in  $Dead(A_1, A_2)$ . Condition (2) says that  $A'_1$  does not share non-autonomous actions in  $EnabRiseDead(A_1, A_2)$  with  $A_2$  if they are enabled by the environment of  $A_{12}$ , may lead inevitably to deadlock states. We claim the following theorem.

**Theorem 5.** *If  $B_1 \sqsubseteq_{B_2}^s B'_1$ , then  $B'_1$  is compatible with  $B_2$  and  $B_1 \parallel B_2 \sqsubseteq B'_1 \parallel B_2$ .*

*Proof.* Under the theorem assumptions, we have to demonstrate that (1)  $Cmp(A'_1, A_2) \neq \emptyset$  and  $\iota_{A'_{12}} \in Cmp(A'_1, A_2)$ , and (2)  $A_{12} \succeq A'_{12}$  and for all  $a \in \Sigma_{A_{12}}^{Im} \cup \Sigma_{A_{12}}^{Om}$ ,  $SemSub_a(B_1 \parallel B_2, B'_1 \parallel B_2)$ ?

(1) First, we prove that  $\iota_{A'_{12}}$  cannot be a deadlock state: if  $\iota_{A'_{12}} \in Dead(A'_1, A_2)$ , then, from  $B_1 \sqsubseteq_{B_2}^s B'_1$  (condition (1) of Definition 14),  $\iota_{A_1} \iota_{A_2} \in Dead(A_1, A_2)$  which is in contradiction with the compatibility of  $B_1$  and  $B_2$ . Therefore, we conclude that  $\iota_{A'_{12}} \notin Dead(A'_1, A_2)$ . Second, we prove that each  $\sigma'$ , starting from  $\iota_{A'_{12}}$  and ending by a deadlock state in  $Dead(A'_1, A_2)$ , contains at least a transition labeled by  $a \in \Sigma_{A'_{12}}^{Im} \cup \Sigma_{A'_{12}}^{Ie}$ : from  $B_1 \sqsubseteq_{B_2}^s B'_1$  (condition (1) of Definition 14), we can deduce that each run  $\sigma'$  starting from  $\iota_{A'_{12}}$  and reaching  $d'_1 d_2 \in Dead(A'_1, A_2)$  is the image of a run  $\sigma$  in  $A_{12}$  starting from  $\iota_{A_{12}}$  and reaching  $d_1 d_2 \in Dead(A_1, A_2)$  such that  $d_1 \succsim d'_1$  or  $d'_1 \in Clos_{A'_1}(c'_1, \Sigma_{A'_1} \setminus \Sigma_{A_1})$  where  $d_1 \succsim c'_1$ :  $\sigma'$  can be decomposed on fragments matching each transition of  $\sigma$ . We consider that  $\sigma = s_0 t_0 [a_0] \dots [a_{n-1}] s_n t_n$  in  $A_{12}$  where  $n \in \mathbb{N}^*$ ,  $s_0 t_0 = \iota_{A_{12}}$ , and  $s_n t_n \in Dead(A_1, A_2)$ . Since  $B_1 \sqsubseteq B'_1$ , we decompose  $\sigma'$  inductively on  $0 \leq l \leq n$ , to fragments  $\sigma'_l$  as follows:

- $l = 0$ :  $\sigma'_0$  is the path of length 0 defined by  $s'_0 t_0 = \iota_{A'_1} \iota_{A_2}$  (we have  $s_0 \succsim s'_0$ );
- $0 < l \leq n$ :  $\sigma'_l$  is the concatenation of  $\sigma'_{l-1}$  with a run  $\alpha'_l$  (matching  $s_{l-1} t_{l-1} [a_{l-1}] s_l t_l$ ) starting from  $s'_{l-1} t_{l-1}$ , reaching  $s'_l t_l$  where  $s_l \succsim s'_l$ , and containing necessarily a transition labeled by  $a_{l-1}$ , and zero or more transitions labeled by autonomous exception-free actions in the set  $\Sigma_{A'_1}^{aut} \setminus \Sigma_{A_1}$ , if  $a_{l-1} \in \Sigma_{A_1} \setminus \Sigma_{A'_1}^e$ , or zero or more transitions labeled by autonomous and exception actions in  $(\Sigma_{A'_1}^{aut} \cup \Sigma_{A'_1}^{Ie}) \setminus \Sigma_{A_1}$  otherwise (the case where  $a_{l-1} \in \Sigma_{A_1}^e$ ).

As  $B_1$  and  $B_2$  are compatible, there is a transition  $s_k t_k [a_k] s_{k+1} t_{k+1}$  in  $\sigma$  such that  $k \in \mathbb{N}_{<n}$  and  $a_k \in \Sigma_{A_{12}}^{Im} \cup \Sigma_{A_{12}}^{Ie}$ . We can observe, based on the previous inductive decomposition, that for the image  $\sigma' = \sigma'_n$  of  $\sigma$ ,  $\Sigma_{A'_{12}}(\sigma')$  contains at least the action  $a_k$  belonging as well to  $\Sigma_{A'_{12}}^{Im} \cup \Sigma_{A'_{12}}^{Ie}$ : from  $B_1 \sqsubseteq_{B_2}^s B'_1$ ,  $a_k \in EnabRiseDead(A_1, A_2)$  is a non autonomous action not shared between  $A'_1$  and  $A_2$  (condition (2) of Definition 14). In addition,  $\sigma'$  reach  $s'_n t_n \in Dead(A'_1, A_2)$  such that  $s_n \succsim s'_n$  or  $s'_n$  is reachable by a fragment of  $\sigma'$  enabling actions in  $\Sigma_{A'_1} \setminus \Sigma_{A_1}$  from a state  $c'_n \in \Upsilon_{A'_1}$  where  $s_n \succsim c'_n$ . Consequently, we deduce that  $\iota_{A'_{12}} \in Cmp(A'_1, A_2)$ .

(2) The condition (1) of Definition 11 is met obviously with  $A = A_{12}$  and  $A' = A'_{12}$ . It remains to prove that there is an expanding simulation  $\succsim'$  from  $A'_{12}$  to  $A_{12}$  such that  $\iota_{A_{12}} \succsim' \iota_{A'_{12}}$ . We take  $\succsim'$  defined by  $\{(s_1 s_2, s'_1 s'_2) \in \Upsilon_{A_{12}} \times \Upsilon_{A'_{12}} \mid s_1 \succsim s'_1 \wedge s_2 \in Cmp(A_1, A_2)\}$ . Finally, it is obvious that for all  $a \in \Sigma_{A_{12}}^{Im} \cup \Sigma_{A_{12}}^{Om}$ ,  $SemSub_a(B_1 \parallel B_2, B'_1 \parallel B_2)$  is true.  $\square$

Given a fourth behavioral contract  $B'_2$  such that  $B'_2$  is composable with  $B'_1$  and  $B_2 \sqsubseteq B'_2$ , the independent implementability property of behavioral contracts is established by the following corollary, which is obviously deductible from theorems 4 and 5.

**Corollary 6.** *If  $B_1 \sqsubseteq_{B_2}^s B'_1$  and  $B_2 \sqsubseteq_{B'_1}^s B'_2$ , then  $B'_1$  and  $B'_2$  are compatible behavioral contracts and  $B_1 \parallel B_2 \sqsubseteq B'_1 \parallel B'_2$ .*

*Proof.* From Theorem 5, we have  $B'_1$  and  $B_2$  are compatible and  $B_1 \parallel B_2 \sqsubseteq B'_1 \parallel B_2$ . We have also, from the corollary premises and Theorem 5,  $B'_1$  and  $B'_2$  are compatible and  $B'_1 \parallel B_2 \sqsubseteq B'_1 \parallel B'_2$ . As  $\sqsubseteq$  is a preorder (Theorem 4), we can conclude that  $B_1 \parallel B_2 \sqsubseteq B'_1 \parallel B'_2$ .  $\square$

Given two interface automata  $A$  and  $A'$ , the refinement relation  $A' \succeq A$  is checkable in time  $\mathbf{O}((|\delta_A| + |\delta_{A'}|) \cdot (|\Upsilon_A| + |\Upsilon_{A'}|))$  [4, 8], where  $|S|$  is the cardinality of a set  $S$ . The algorithm of checking refinement between interface automata, in our approach, can be deduced naturally

form that proposed in [7]. Safe refinement can be checked in linear time by forward or backward traversals, that is  $B_1 \sqsupseteq_{B_2}^s B'_1$  can be checked in time  $\mathbf{O}(|\delta_{(B_1|B_2)}.\mathcal{A}| \cdot |\delta_{(B'_1|B_2)}.\mathcal{A}|)$ . The previous complexity is extended by the satisfiability decision problems related to the semantic substitutability conditions of common observable method actions between the refinement of a behavioral contract and its abstraction.

## 6 Conclusions

This report is a contribution to the design of object-oriented component-based applications using behavioral contracts. This formalism combines protocol and semantic levels of component interface specifications. The protocol level is designed by means of interface automata and the semantic level is defined on methods by pre/postconditions and specifications stated on their parameters and components attributes. The optimistic approach of interface automata composition is accordingly adapted to fulfill the interaction aspects between components in the object-oriented context. Refinement of behavioral contracts is defined from the perspective of OOCBD. It is based on a simulation relation allowing addition of refinement details about the behavioral protocol and semantics of provided services common to refined and abstract versions of a component behavioral contract. The work is illustrated by a design case study of trains protection functions in railway CBTC systems.

## References

- [1] IEEE Std. 1474.1-1999 for Communications-Based Train Control (CBTC) Performance and Functional Requirements (rev. 2004). pages 1–45.
- [2] Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, CENELEC EN 50128. 2001 (rev. 2011).
- [3] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, NY, USA, 1996.
- [4] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *Procs. of the 9th Int. Conf. on Concurrency Theory*, pages 163–178, London, UK, 1998. Springer-Verlag.
- [5] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer Berlin Heidelberg, 1991.
- [6] S. Chouali, H. Mountassir, and S. Mouelhi. An I/O automata-based approach to verify component compatibility: application to the CyCab car. *Elec. Notes Theor. Comput. Sci.*, 238:3–13, June 2010.
- [7] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.
- [8] L. de Alfaro and T. A. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

- [9] European Railway Agency. ERTMS/ETCS Functional Requirements Specification. Technical report, 2010.
- [10] European Railway Agency. ERTMS/ETCS System Requirements Specification. Technical report, 2010.
- [11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [12] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841, November 1994.
- [13] J. Mikác and P. Caspi. Temporal refinement for Lustre. In *Procs. of Languages Applications and Programming, SLAP’05*, Elec. Notes Theor. Comput. Sci. Elsevier, 2005.
- [14] S. Mouelhi, S. Chouali, and H. Mountassir. Refinement of interface automata strengthened by action semantics. *Elec. Notes Theor. Comput. Sci.*, 253:111–126, October 2009.
- [15] W. Schön, G. Larraufie, G. Moens, and J. Pore. *Railway Signalling and Automation*, volume 3. La Vie du Rail, 2013.
- [16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.





---

FEMTO-ST INSTITUTE, headquarters

32 avenue de l'Observatoire - F-25044 Besançon Cedex FRANCE

Tél : (33 3) 81 85 39 99 – Fax : (33 3) 81 85 39 68 – e-mail: [contact@femto-st.fr](mailto:contact@femto-st.fr)

FEMTO-ST - AS2M : TEMIS, 24 rue Alain Savary, F-25000 Besançon

FEMTO-ST - DISC : UFR Sciences - Route de Gray - F-25030 Besançon cedex France

FEMTO-ST - ENERGIE : Parc Technologique, 2 Av. Jean Moulin, Rue des entrepreneurs, F-90000 Belfort France

FEMTO-ST - MEC'APPLI : 24, chemin de l'épitahe - F-25000 Besançon France

FEMTO-ST - MN2S : 32, rue de l'Observatoire - F-25044 Besançon cedex France

FEMTO-ST - OPTIQUE : UFR Sciences - Route de Gray - F-25030 Besançon cedex France

FEMTO-ST - TEMPS-FREQUENCE : 26, Chemin de l'Epitahe - F-25030 Besançon cedex France

---

<http://femto-st.fr>